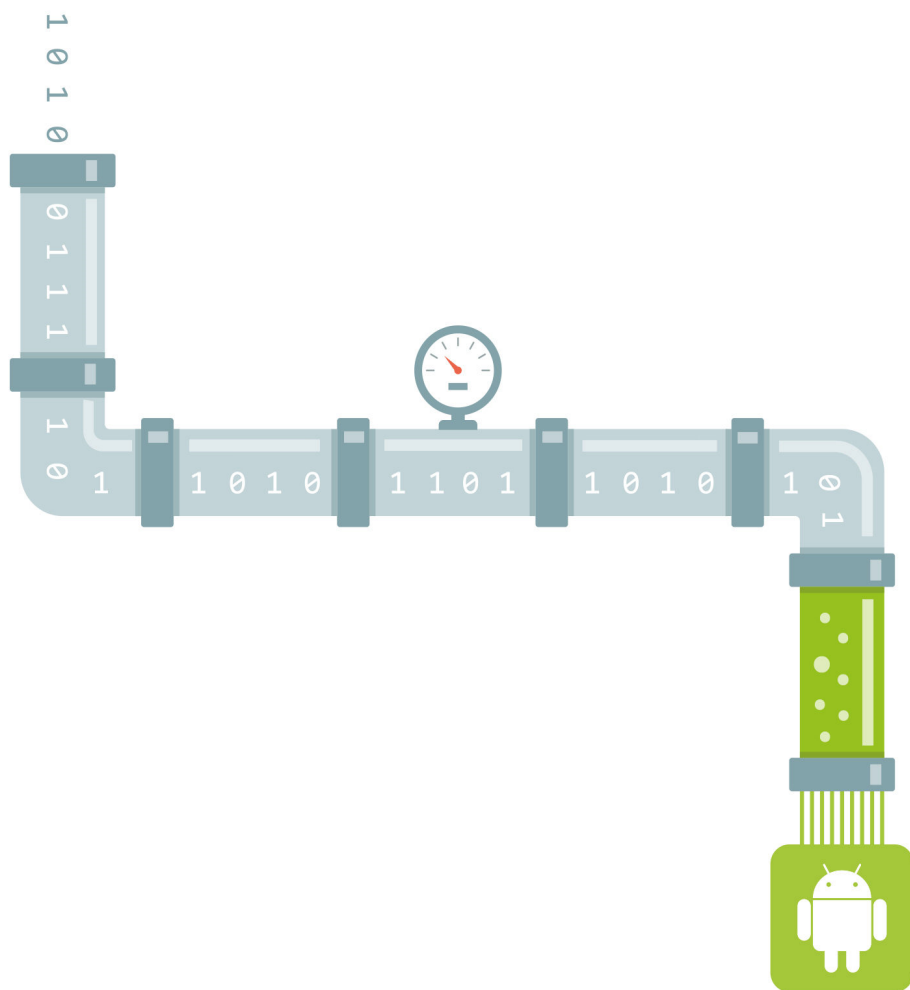


Entrega contínua em Android

Como automatizar a distribuição de apps





ISBN

Impresso e PDF: 978-85-5519-219-7

EPUB: 978-85-5519-220-3

MOBI: 978-85-5519-221-0

Você pode discutir sobre este livro no Fórum da Casa do Código: <http://forum.casadocodigo.com.br/>.

Caso você deseje submeter alguma errata ou sugestão, acesse <http://erratas.casadocodigo.com.br>.

AGRADECIMENTOS

Agradeço à minha bisavó, Tereza (*in memoriam*), e à minha avó, Iara, por estarem ao meu lado desde o meu primeiro dia de vida e por me permitirem ser hoje um homem de bom caráter, com dignidade e ter sido premiado com uma educação de excelente qualidade. Amo vocês!

Também gostaria de agradecer ao Instituto de Informática da Universidade Federal do Rio Grande do Sul (UFRGS), pela excelente qualidade de ensino fornecida a mim durante a graduação.

Um agradecimento especial à Casa do Código. Ao Adriano Almeida, por permitir a publicação deste material, e para a Vivian Matsui, por me acompanhar nesta jornada com suas revisões e sugestões para que fizéssemos um livro de boa qualidade. Muito obrigado a vocês todos.

Por fim, gostaria de agradecer a você, prezado leitor, por ter adquirido este livro. Ele foi resultado de muita pesquisa, experimentos e revisões, de modo a lhe fornecer um material diferenciado e que possa ser útil em sua vida profissional.

SOBRE O AUTOR



Figura 1: Roger Silva

Desde 2012 com atuação no setor de TI, é Engenheiro de Software com forte experiência em desenvolvimento mobile. Já trabalhou também com desenvolvimento front-end, back-end e em manutenção de software legado. Oficialmente certificado Scrum Master (pela Scrum Alliance), é apaixonado por trabalhar com todas as correntes do Agile — Scrum, Lean, Kanban e XP.

Bacharel em Ciência da Computação pela Universidade Federal do Rio Grande do Sul (UFRGS), é também blogueiro e palestrante em eventos sobre tecnologia. Nas horas vagas, é frequentador assíduo do estádio do seu time de coração, game maníaco desde os quatro anos de idade e viciado em happy hours.

Blog: <http://www.orogersilva.com/>

LinkedIn: <https://br.linkedin.com/in/orogersilva>

PREFÁCIO

Sua empresa será a responsável por desenvolver um novo app Android para seu cliente. O arquiteto de software será o encarregado por dar o *kick-off* do projeto. Ele abre o Android Studio, cria um novo projeto, nomeia-o, redefine seu nome de pacote, e define a API mínima suportada pelo app e plataformas-alvo (telefone, tablet, Android Wear, TV, Android Auto etc). *Então, é hora de construir a aplicação!* Na verdade, não. É hora de consertá-la. Sim. Consertá-la. Pois é tido como premissa que todo novo software está quebrado até que ele seja validado.

Em contrapartida ao modelo *Waterfall*, em que um novo software tem suas funcionalidades somente validadas ao final do ciclo de seu desenvolvimento, quem sabe validá-lo desde a escrita de suas primeiras linhas de código?

Mas até termos um protótipo funcional levará dias. O que validaremos então? A resposta é simples: suas estruturas internas. Desde o primeiro dia de desenvolvimento, se possível.

De métodos de instância até contextos funcionais em nível de *business*, o app terá sua chance de ser validado. E não somente isso. Há a geração de relatórios resultantes de análises de suas estruturas e publicação automatizada para o Google Play. Isso desde o primeiro dia de implementação. Várias vezes ao dia, caso desejado.

Esse contexto expressa o dia a dia de um time de desenvolvimento que faz **entrega contínua** de seu software. Validações sendo realizadas o mais brevemente possível, publicações do app para o cliente diversas vezes durante a semana, e *feedback* antecipado sobre o estado atual do app da parte do cliente. Isso não é sonho. Isso é realidade.

Desde a publicação do livro *Continuous Delivery — Reliable Software Releases Through Build, Test, And Deployment Automation*, de Jez Humble e David Farley, muito foi discutido sobre o assunto. Porém, um vácuo literário ainda persistia.

Como fazer entrega contínua de aplicações mobile? Mais especificamente, apps Android. Trata-se de um contexto peculiar. Apps são publicados para o Google Play (ou para outras plataformas de distribuição). Alguns tipos de testes requerem um dispositivo para suas execuções. Assim, a comunidade de desenvolvimento mobile merecia uma publicação como esta para solucionar essas questões.

A quem este livro se destina?

Você é desenvolvedor mobile Android? Esta publicação será muito bem-vinda a você. Você é desenvolvedor mobile iOS? Arrisco-me a dizer que também lhe será muito útil. Pois muitas ideias retratadas aqui são compartilhadas entre as duas plataformas mobile. O que diferirá serão as tecnologias.

Você é profissional de TI, não necessariamente um desenvolvedor? Este livro também fará de você um profissional melhor, pois, além da parte teórica sobre os conceitos de integração, entrega e deployment contínuo, serão expostos como esses conceitos são colocados em prática.

Porém, esta publicação é direcionada não somente para profissionais, como também para estudantes. Apesar de não ser voltada para calouros, o contato com o conteúdo deste livro permite com que esse público esteja a par do que o mercado de produção de software de alta qualidade demanda de seus profissionais.

O único requisito técnico recomendado para a leitura deste livro é o conhecimento básico de programação em Java, tal que facilite a

compreensão do leitor sobre o funcionamento de testes automatizados para apps Android.

Sumário

1 Primeiros passos e definições	1
1.1 O problema	2
1.2 Pipeline de deployment	4
1.3 Integração contínua	13
1.4 Entrega contínua x Deployment contínuo	16
1.5 O caso de estudo	18
2 Gerenciamento de branches	21
2.1 Gerenciando branches em um contexto com integração contínua	22
2.2 Estratégias de branching	24
3 Testes automatizados	35
3.1 O que é um teste automatizado?	36
3.2 Testes unitários	43
3.3 Análise estática de código	51
3.4 Testes de integração	60
3.5 Testes de integração em Android	63
3.6 Testes funcionais	71
4 Ferramentas para integração e entrega contínua	77
4.1 Travis CI	79

4.2 GoCD	86
4.3 Jenkins	98
4.4 Comparação entre ferramentas	122
4.5 Publicação no Google Play	127
5 Distribuições over-the-air	140
5.1 O conceito	140
5.2 Requisitos para atualizações OTA	143
5.3 HockeyApp	144
5.4 Crashlytics	157
5.5 Conclusão	169
6 Bibliografia	171

PRIMEIROS PASSOS E DEFINIÇÕES

Concepção, projeto, desenvolvimento, testes, todas essas etapas são algumas das fases que constituem a construção de um novo software. A literatura que aborda essas fases é vasta e em constante atualização de acordo com novas ideias e tecnologias que surgem com o passar dos dias.

Porém, a fase determinante durante o ciclo de vida de aplicação sofre pela escassez de literatura, e é de suma importância para a qualidade e manutenção do software sob desenvolvimento: a fase de entrega. A escolha pela forma de entrega do software ao(s) cliente(s) é fundamental. Mas o que é a entrega? *Pode ser empacotar o software, armazená-lo em um disco e enviar ao cliente em um envelope pelo correio?* Sim, pode! *E enviá-lo por e-mail também vale?* Claro! Porém, são mecanismos arcaicos (para não dizer coisa pior). Existem formas muito mais adequadas e corretas.

Se já não bastasse a escassez literária sobre entrega de software, o cenário é ainda mais obscuro quando falamos sobre entrega de software mobile. O contexto de entrega nesse cenário tem suas particularidades. Ainda mais se considerarmos que a fase de entrega pode ser dependente de outros elementos, tais como a estratégia de gerenciamento de *branches* no controle de versão e a execução de testes automatizados.

Além disso, o aplicativo será publicado onde? No Google Play? Em um ambiente privado? Todas essas escolhas podem afetar a entrega do software. E a entrega de aplicativos mobile (especialmente aplicativos Android) também merece sua devida atenção.

1.1 O PROBLEMA

Ao iniciar o projeto para o desenvolvimento de um novo software, o time de desenvolvimento reúne-se em uma *sala de guerra* para discutir questões como requisitos do novo sistema, tecnologias a serem usadas, metodologias de desenvolvimento, dentre outras questões. Do mesmo modo, esse software pode ser construído por um desenvolvedor autônomo, tal que ele terá de organizar seu dia a dia de alguma forma (como através de uma simples planilha), para que, uma vez o software pronto (ou, ao menos, um conjunto de funcionalidades prontas), possa entregá-lo ao cliente.

Independente de ser um desenvolvedor freelancer ou um time de desenvolvimento de software já maduro, o responsável pelo desenvolvimento desse novo software deve dar atenção especial a um item, entre outros já citados: a entrega do software ao cliente.

Dentre as perguntas a serem feitas sobre a fase de entrega estão:

- Como será realizada a entrega?
- Com qual frequência entregar?
- Será adotado algum processo automatizado ou a entrega do software será manual?
- Como o software é validado antes da entrega?

Quais são as respostas corretas para essas perguntas? Corretas não existem. Porém, existem boas práticas.

Por exemplo, um processo de entrega automatizado evita a falha humana, uma vez que esta é possível de ocorrer durante o gerenciamento da configuração, preparação dos dados ou escolha dos passos para construção do artefato de software. Realizar entregas frequentes de funcionalidades, tão logo estejam implementadas, permite o imediato feedback do cliente, de modo que melhorias possam ser realizadas para a próxima entrega. Validar o software através de testes automatizados é primordial (apesar de testes manuais exploratórios terem também a sua importância). Porém, não é simples.

De modo a desmistificar a automatização de testes sobre aplicativos Android, o *capítulo 3* desta publicação tratará mais sobre cada categoria de testes, e as ferramentas utilizadas para a implementação dos testes pertencentes a cada uma delas.

A entrega de um aplicativo mobile

A obra tida como referência sobre o assunto é o livro *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, de Jez Humble e David Farley. Apesar de ser um excelente livro, ele não retrata alguns dos problemas com que nos deparamos ao entregar software para plataformas mobile.

Por exemplo, como automatizar a distribuição do app para o Google Play? Como executar testes automatizados sobre as funcionalidades do app, sendo que eles necessitam de interação com a interface gráfica e, portanto, é necessário um dispositivo físico ou um emulador? Como disponibilizar o app para a equipe de testes, em vários dispositivos, de modo que o app possa ser validado manualmente? E o principal, como tratar todos esses requisitos de modo que, quando um desenvolvedor comitar seu código-fonte para um repositório de código remoto, uma bateria de testes automatizados seja executada, acompanhada por uma análise de

cobertura de código-fonte, verificações de regras de negócios e, por fim, pela distribuição automatizada do app para o Google Play? São respostas para esses tipos de problemas que este livro traz.

1.2 PIPELINE DE DEPLOYMENT

O **pipeline de deployment** é uma implementação automatizada dos processos de build, deploy, testes e release de um software.

O **BUILD** é o processo de construção do software, efetuado através de arquivos de códigos-fonte, bibliotecas e outros artefatos de software auxiliares. Já um **deploy** é a geração resultante de um processo de build, ou seja, o artefato do software construído.

Os **testes** são as validações realizadas sobre o artefato de software de forma automatizada e não automatizada. Por fim, o processo de **release** é a disponibilização do software ao público geral ou a um grupo de usuários pré-selecionados.

Ou seja, por um pipeline de deployment, uma vez que um desenvolvedor comitar alterações do código-fonte de uma aplicação para o repositório de código, ao final do pipeline poderá estar disponível uma versão da aplicação validada por seus testes e, até, ser entregue ao(s) cliente(s) de forma automatizada. Um possível pipeline de deployment é mostrado a seguir:

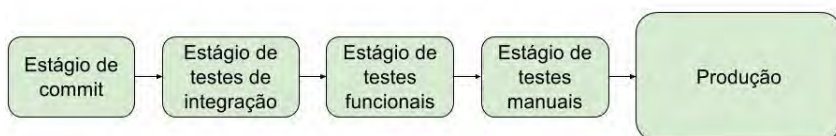


Figura 1.1: Pipeline de deployment

Estágio de commit

Assim que há um commit para o repositório da aplicação, o pipeline de deployment pode ser notificado sobre alterações na aplicação trabalhada. Feita essa notificação, o código-fonte, assim como os demais recursos que compõem a aplicação, são direcionados ao primeiro estágio do pipeline deployment, também chamado de **estágio de commit**. Esse estágio, é composto pelas seguintes subetapas:



Figura 1.2: Estágio de commit

A fase de **compilação** aqui pode ser entendida como o início do processo de *build* (construção) da aplicação. E por que início? Ao executar o build para um aplicativo Android, o Gradle (sistema de build usado para construir apps Android) é formado por passos de build. Dentre os passos que compõem o build como um todo, estão aqueles responsáveis por executar os testes unitários, análise estática de código e empacotamento da aplicação. Sendo assim, todas as subetapas citadas definem o build como um todo.

A próxima etapa executa **testes unitários** contra a aplicação.

Esses são testes que requerem pouco tempo para sua execução, fornecendo, assim, rápido feedback aos responsáveis pelo desenvolvimento da aplicação (mais adiante, neste livro, testes unitários serão explicados mais detalhadamente). Segue uma suíte com testes unitários validando funcionalidades de componentes internos de uma aplicação Android:

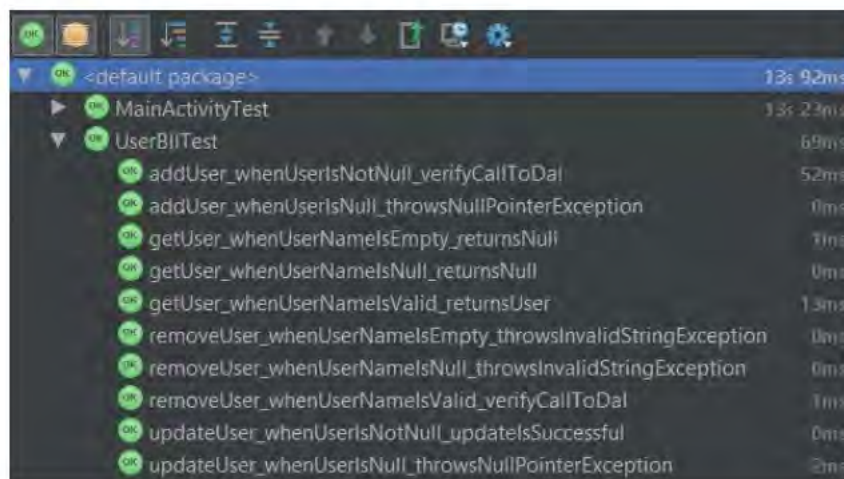


Figura 1.3: Suíte de testes unitários

Em seguida, a aplicação é encaminhada para a subetapa de **análise estática de código**. Essa fase trata de analisar o código-fonte da aplicação e reportar possíveis *bad smells* no código que podem influenciar negativamente em outros pontos da aplicação. Dentre os problemas, podem ser citados:

- Código-fonte não segue uma convenção de escrita;
- Baixa cobertura de código;
- Potenciais bugs;
- Issues de segurança.

A seguir, veja um trecho de código-fonte no qual não é seguida a regra em que os caracteres `<` e `>` são concatenados ao argumento

de tipo de um tipo genérico:

```
ArrayList< Friend > friends = new ArrayList<>();
```

Figura 1.4: Convenção de código não adotada

Após executar a ferramenta de análise de código, é reportado o seguinte resultado:

File C:\Git\RachaContalapp\src\main\java\com\lorogersilva\rachacontalview\activity\DeskActivity.java	
Error Description	
'<' é seguido de espaço em branco.	
'>' é precedido por espaço em branco.	

Figura 1.5: Resultado da análise estática de código

Existem muitos outros pontos de atenção ao final de uma análise como essa. *Mas afinal, essa é uma subetapa crítica?* Depende. Depende dos requisitos da aplicação a ser construída.

Uma análise de código automatizada não resolve completamente o problema, mas serve como um ótimo suporte ao alcance desse objetivo. A aplicação é basicamente um MVP (*Minimum Viable Product*), não é crítica e contém funcionalidades muito triviais? Talvez uma análise estática de código não valha o esforço nesse caso. Logo, essa subetapa, dependendo do contexto, não é obrigatória dentro de um pipeline de deployment. Mas é sempre bem-vinda.

Por fim, a subetapa de **empacotamento** trata de agrupar todos os componentes resultantes do processo de build e gerar o arquivo do aplicativo Android, formado pela extensão `apk`. A execução de validações durante o estágio de commit não requer um emulador Android. Isso é muito importante.

O principal motivo para um processo de entrega contínua ser

dividido em etapas através de um pipeline de deployment é viabilizar rápido *feedback*. Testes automatizados no estágio de commit são rápidos (preferencialmente, não podem levar mais do que dez minutos para serem finalizados para uma grande aplicação). Essa rapidez viabiliza ao time de desenvolvimento (e outros *stakeholders*) ser notificado de que há algo errado e, se for o caso, todos devem trabalhar juntos para que o build seja estabilizado.

Estágio de testes de integração

A saída do estágio de commit (o arquivo executável do aplicativo Android) estará disponível em uma pasta dentro do ambiente do pipeline. Uma vez disponível, ele será usado como entrada para o próximo estágio que, no pipeline esquematizado neste capítulo, é o **estágio de testes de integração**.

Como o próprio nome diz, esse é um estágio responsável por testar a integração. *Mas integração de quê?* A integração dos componentes que compõem o aplicativo Android. Em outras palavras, será verificado se as interações entre os componentes do aplicativo, dado um cenário e dados de entrada, resultam em um comportamento esperado.

Por exemplo, um teste de integração viável é verificar se o componente responsável por gerenciar a persistência de informações na base de dados interage de forma correta e esperada com essa, como mostrada a seguir:

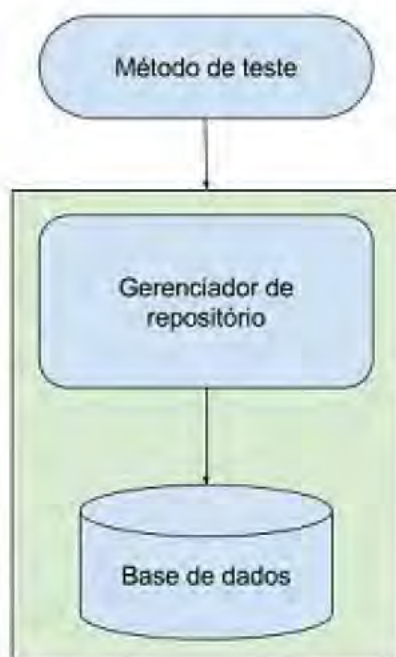


Figura 1.6: Diagrama de teste de integração

Os mais diversos componentes de uma aplicação Android podem ser validados por essa classe de testes, dentre eles:

- Componentes de acesso a dados;
- Gerenciadores de acesso à rede;
- Componentes de acesso ao sistema de arquivos.

Testes de integração acessam componentes dependentes de dispositivo físico ou emulador. Logo, devido ao tempo de acesso e comunicação a componentes externos à aplicação, é natural o estágio de testes de integração levar mais tempo para finalizar sua execução em relação ao estágio de commit. Mais detalhes sobre testes de integração serão detalhados em capítulos seguintes.

Estágio de testes funcionais

O próximo estágio do pipeline de deployment caracteriza-se por tratar daquilo que o sistema como um todo faz. Ou seja, os casos de teste pertencentes a esse estágio são baseados nas especificações da aplicação, por vezes definidas por um *product owner* ou um membro do time de *quality assurance* (QA). Os testes automatizados pertencentes a essa classe são chamados de **testes funcionais**.

Devido ao fato de os critérios de aceitação de cada caso de uso poderem ser especificados com a abordagem BDD (*Behavior Driven Development*), é uma boa prática usar a descrição de cada critério de aceitação para nomear cada método de teste funcional:

BDD não será abordado neste livro, mas sugiro uma excelente referência, o livro *User Stories Applied: For Agile Software Development*, de Mike Cohn (2004).

```
/**
 * CENÁRIO: Adiciona novo amigo em lista de amigos vazia
 * DADO que não existem amigos ainda
 * QUANDO eu adiciono um novo amigo
 * ENTÃO uma nova linha é mostrada na lista
 */
```

Figura 1.7: Critério de aceitação

```
@Test
public void dadoQueNaoExistemAmigosAinda_quandoEuAdicionoUmNovoA
migo_entaoUmaNovaLinhaEhMostradaNaLista() {
    // Implementação do método de teste funcional deve vir aqui
}
```

Entretanto, devemos ter bom senso se essa prática de nomeação

for levada ao pé da letra. No exemplo mostrado, o nome do método é muito longo. No entanto, o objetivo principal de qualquer nome de método é expressar claramente sua responsabilidade. Como trata-se de um método de teste, o comprimento do nome do método é ainda válido, pois, após a execução de uma suíte de testes automatizados, em caso de falha na execução de um método de teste, devido ao seu nome ser autoexplicativo, é instantânea a descoberta do contexto sob o qual o teste falhou. Porém, não devemos exacerbar. É necessário bom senso sobre a decisão em escolher usar essa prática de nomeação ou não.

A execução de testes funcionais em Android requer o uso de um emulador e, por isso, são também chamados de testes peso-pesado devido ao fato de poderem acessar recursos externos à aplicação (como base de dados, chamadas a APIs, comunicação interprocessos), aos longos fluxos de execução de cada método de teste. Afinal, cada método corresponderá a interação do usuário com a aplicação através de um cenário de caso de uso. Claro que também são chamados assim devido a uma possível lentidão do emulador Android. Logo, testes funcionais são caros, portanto, é preferível implementá-los em menor escala em relação a testes unitários, por exemplo.

Estágio de testes manuais

Finalizada mais uma etapa do pipeline de deployment, é dado início a outra etapa, não automatizada, na qual os membros da equipe de testes atuam sobre a aplicação. No estágio de **testes manuais**, testadores podem realizar testes exploratórios, testes de *user experience*, ou mesmo a validação dos critérios de aceitação de casos de uso já validados no estágio de testes funcionais, com o adicional de estressar a aplicação em cenários mais complexos.

Por vezes, equipes responsáveis pela construção de softwares

preferem suprimir esse estágio no processo de desenvolvimento com o argumento de que automatizar todo processo de testes através de um pipeline torna desnecessária a presença humana para validações manuais. Isso é um erro. Testadores têm a visão e habilidades que desenvolvedores não possuem. Eles sabem julgar e explorar prováveis fluxos de execução que resultariam em erros na execução da aplicação.

Produção

O ambiente de produção mais conhecido para aplicativos Android é o Google Play. Ou seja, é a loja oficial para publicação e disponibilização de aplicativos da plataforma Android. Porém, não é o único.

Um aplicativo Android, após a passagem pelo estágio de testes manuais, pode ser despachado para dispositivos particulares de usuários pré-selecionados. Esse tipo de distribuição é conhecida como *over-the-air*. Como exemplo de plataformas desse tipo, podemos citar o *Crashlytics* e o *HockeyApp*, que serão abordados nos capítulos seguintes.

O app também pode distribuído através do site da companhia proprietária da aplicação. O WhatsApp adota essa prática, a qual permite a seus usuários realizarem o download de versões beta da aplicação ainda não publicadas no Google Play.



Figura 1.8: Distribuição de aplicativo no site

Tal qual faz o WhatsApp, é possível distribuir um app para um grupo de usuários pré-selecionados através do Google Play Developer Console, plataforma para gerenciamento de apps publicados no Google Play e de suas informações. Esse tipo de distribuição, também conhecida como **distribuição beta**, precede a publicação do app no Google Play ao público geral, de modo que feedback sobre o app possa ser coletado de um grupo de usuários, o app melhorado e, em seguida, distribuído oficialmente no Google Play.

1.3 INTEGRAÇÃO CONTÍNUA

Ao construir um novo software, ele estará quebrado até que se prove o contrário. Não é recomendado validar funcionalidades da aplicação somente quando elas estiverem próximas de estarem prontas, pois postergar a fase de testes encarece o desenvolvimento da aplicação e de sua manutenção em longo prazo. Isso se deve ao

fato de muitas das estruturas da aplicação já terem sido implementadas até sua quase finalização. Um teste que encontra uma falha no contexto em que a fase de testes é postergada para o final do ciclo de desenvolvimento torna modificações mais trabalhosas de serem realizadas.

Uma boa estratégia para demonstrar a corretude de uma aplicação desde o início de seu ciclo de desenvolvimento é através de testes unitários. Isso porque eles atuam sobre componentes individuais que formam a arquitetura da aplicação de forma que o software será devidamente validado desde suas menores estruturas até as maiores.

Acompanhada de uma suíte inicial de testes unitários, para manter o histórico de modificações sobre o software sendo construído (dentre outras utilidades), é necessário o uso de uma ferramenta de controle versão. Digamos que durante o desenvolvimento da nova aplicação uma falha seja introduzida (não propositalmente) pelo desenvolvedor, falha esta que não ocorria anteriormente.

Logo, um ou mais dos testes unitários falharão. Existem duas atitudes a serem tomadas nesse caso: ou é realizada manutenção sobre o código da aplicação, de modo que a regressão ocorrida seja removida; ou, em caso de urgência, pode ser realizado um rollback do código-fonte. Através de um sistema de controle de versão, o rollback pode ser facilmente realizado, de forma controlada e registrada.

Um build automatizado e executado em um curto período de tempo permite que o time de desenvolvimento *comite* alterações da aplicação-alvo, de modo que sempre obtenha feedback sobre possíveis regressões rapidamente, para que a aplicação possa ser consertada também mais rapidamente que o habitual.

Comitando modificações sobre o software nesse contexto, tão continuamente quanto possível, com o apoio de um sistema de controle de versão, um build automatizado ea concordância do time em trabalhar sob esse regime viabilizam a prática da **integração contínua** (*continuous integration*).

Diversos são os softwares que facilitam a adoção de integração contínua por parte de times de desenvolvimento. Tais softwares podem ser chamados de ferramentas de integração contínua. Dentre as ferramentas disponíveis podem ser citadas o Travis CI, Circle CI, Jenkins, dentre outras.



Figura 1.9: Travis CI



Figura 1.10: Circle CI



Figura 1.11: Jenkins

Cada ferramenta contém suas particularidades e cada uma será devidamente explorada mais adiante neste livro.

1.4 ENTREGA CONTÍNUA X DEPLOYMENT CONTÍNUO

A prática de integração contínua pode ser estendida à etapa de entrega do software ao cliente. Por meio de um pipeline de deployment, o time de desenvolvimento demonstra confiança em entregar o software construído para o cliente a qualquer momento, uma vez que o código-fonte da aplicação tenha sido integrado e validado por seus testes automatizados diversas vezes durante o ciclo de desenvolvimento.

Para que a entrega se concretize, sob esse contexto, bastam apenas duas ações por parte do desenvolvedor:

- Definir a nova versão da aplicação a ser entregue;
- Executar o push no sistema de controle de versão através de sua estação de trabalho.

No desenvolvimento de um aplicativo Android, ao final do

processo de entrega, o app estará disponível no Google play para que usuários possam baixá-lo em seus dispositivos móveis.

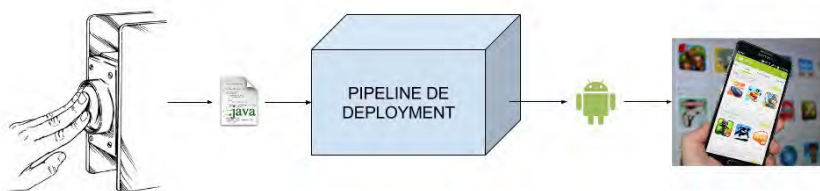


Figura 1.12: Deployment "push-button"

Sob essas condições, a **entrega contínua** (*continuous delivery*) ocorre uma vez que um time de desenvolvimento assegura que possam ser comitadas alterações sobre o software e, após validadas por seus testes, entregues tão frequentemente quanto possível para o ambiente de produção (em Android, para o Google Play) a qualquer momento. Porém, o processo não ocorre em sua totalidade sempre de forma automatizada, como mostra a figura a seguir:

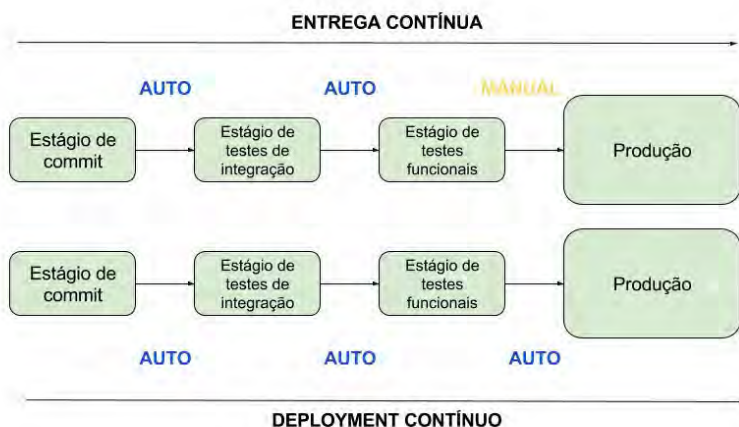


Figura 1.13: Entrega contínua x Deployment contínuo

A entrega contínua ocorre quando a movimentação do artefato de software gerado para o estágio de produção é realizada manualmente, através de um pipeline de deployment. *E por que realizar esse último movimento para o estágio de produção de forma manual, se um mecanismo automatizado pode cuidar disso?*

Decisões de negócios podem requerer esse movimento manualmente. Funcionalidades podem ser incrementadas ao software sob construção. Porém, somente ao ser atingido um conjunto de funcionalidades já esperadas pelo cliente (por exemplo, ao final de um sprint), torna-se interessante a entrega e, por consequência, a aplicação pode ser movida manualmente para o ambiente de produção. Porém, caso esse movimento para o estágio de produção ocorra de forma automatizada, é dito que o software está sob um ambiente apto para **deployment contínuo** (*continuous deployment*).

1.5 O CASO DE ESTUDO

Ok! Após vários conceitos citados até então, tais como:

- Integração contínua
- Entrega contínua
- Deployment contínuo
- Testes unitários
- Testes de integração
- Análise estática de código
- Testes funcionais
- Pipeline de deployment

A pergunta que se faz é: *Como eu faço entrega contínua na prática para um app Android?*

Para demonstrar como entrega contínua é implantada no

mundo real através deste livro, será usado como exemplo um app Android com nome de **Racha Conta**. Seguem telas do aplicativo:

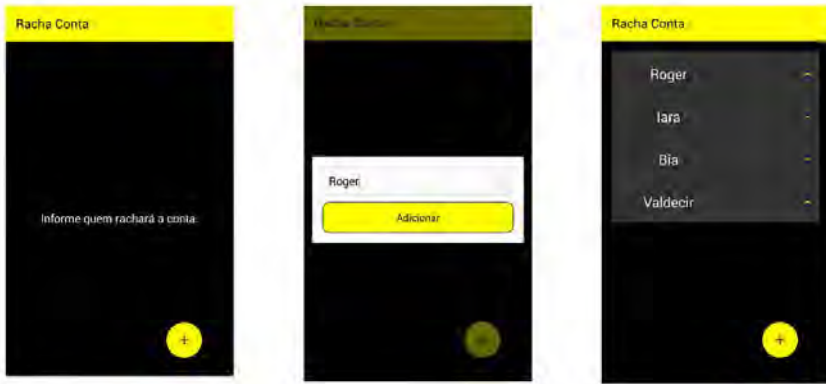


Figura 1.14: Racha Conta

O aplicativo auxilia o usuário a dividir a conta em um encontro entre amigos. Mas isso não é importante. O que realmente importa é o fato desse app ter sido desenvolvido desde sua concepção com o auxílio de um pipeline de deployment.

A existência de uma suíte de testes dos mais variados tipos possibilita com que o app escale arquiteturalmente, se necessário, de modo que um alto número de modificações no código-fonte da aplicação não introduza regressões. Além disso, o pipeline de deployment foi projetado de forma a permitir deployment contínuo. Ou seja, o desenvolvedor executa o comando `git push...` em sua máquina e, após alguns minutos, uma nova versão do Racha Conta estará publicada no Google Play.

O código-fonte do app Android Racha Conta pode ser encontrado no GitHub: <https://github.com/orogersilva/racha-conta-android>.

GERENCIAMENTO DE BRANCHES

Um dos elementos-chave para a prática de integração contínua, como já mencionado no capítulo anterior, é a adoção de um sistema de controle de versão. Além de manter o histórico da evolução do software sob desenvolvimento, é importante também definir como as alterações do software serão gerenciadas antes de serem comitadas para o repositório de código-fonte. Isso porque impacta diretamente na frequência com que o software sob desenvolvimento será integrado.

Este livro usará como base o sistema de controle de versão distribuído Git, assim como o serviço de hospedagem de projetos compartilhados GitHub, que faz uso e oferece suporte a funcionalidades do Git, para exemplificar o gerenciamento de alterações de um projeto de aplicativo Android.

Apesar de não serem tratadas funcionalidades do Git e GitHub neste livro, tendo em vista que o principal escopo dele é a prática de entrega contínua, você pode obter mais informações sobre eles:

- Git: <https://git-scm.com/>
- GitHub: <https://github.com/>

2.1 GERENCIANDO BRANCHES EM UM CONTEXTO COM INTEGRAÇÃO CONTÍNUA

A adoção da prática de integração contínua recomenda que todos os artefatos de um aplicativo em desenvolvimento sejam comitados em uma única branch, mais especificamente na branch *master*. Isso porque o software sob essa branch será continuamente integrado após cada commit de cada desenvolvedor.

Assim, o software estará sempre em um estado *releasable*. Ou seja, após passar por todos os estágios de um pipeline de integração, estará pronto para ser entregue ao cliente a qualquer momento.

A integração contínua é uma prática. Requer muita disciplina da parte do time de desenvolvimento. E, principalmente, o time precisa ter consciência do seguinte fato: caso o build no servidor de integração quebrar em decorrência de um commit, o time deve voltar todos os seus esforços em consertá-lo, para, somente após esse conserto, retomar o processo de desenvolvimento de novas features normalmente.

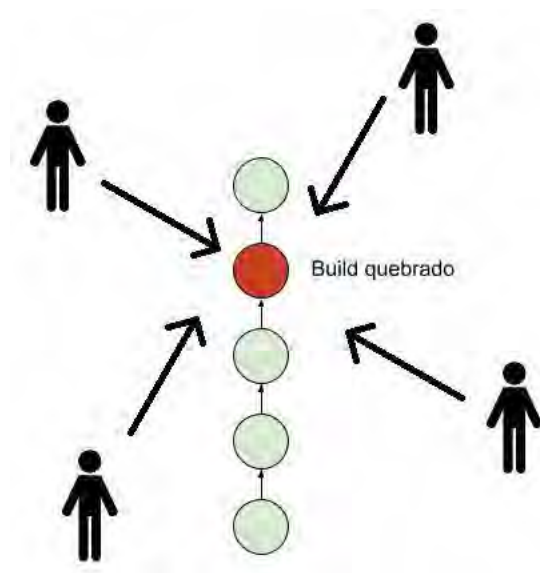


Figura 2.1: Build quebrado

A falta de experiência de alguns membros do time pode levar a quebras frequentes do build, tornando o estado do software não *releasable*. Isso faz parte do processo. Porém, uma vez quebrado, todos no time, se necessário, trabalharão para que a integração seja estabilizada. A maturidade do time e qualidade do software desenvolvido dependem dessa postura.

A escolha pela adoção de múltiplas branches é viável. Porém, o software não estará sob o contexto de integração contínua, pois, uma vez que uma nova branch for criada a partir da branch *master*, seu merge para a linha de desenvolvimento principal pode levar muito tempo. Em consequência, isso resulta em uma chance alta de conflitos no merge quando ele for realizado e, portanto, em uma chance muita alta de quebra no build.

Dessa forma, a dificuldade de conserto do build será maior, uma vez que a feature trabalhada na branch dedicada levou muito tempo para ser integrada à linha principal de desenvolvimento.

Porém, deve-se levar em consideração que cada contexto de desenvolvimento tem as suas necessidades. Adaptações devem ser adotadas para supri-las. Por vezes, um app deve ser desenvolvido com o uso de múltiplas branches. Assim, seguem possíveis abordagens para manter o controle sobre o histórico do software em desenvolvimento.

2.2 ESTRATÉGIAS DE BRANCHING

A seguir, são descritas duas estratégias de gerenciamento de branches para manter o histórico de um software durante seu ciclo de vida. Será dada prioridade a essas duas estratégias por serem simples, predominantes durante a manutenção de projetos de software e facilitarem a configuração de pipelines de deployment.

Desenvolvendo na master

O sistema de controle de versão conterá uma única branch: a *master*. O código comitado por todos os desenvolvedores estará sobre a *master*.

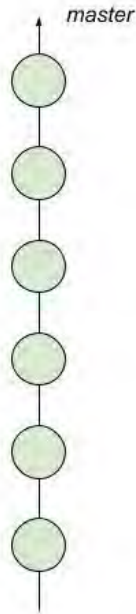


Figura 2.2: Comitando na branch master

Trata-se da abordagem mais recomendada para a adoção da prática de integração contínua. Isso porque, após cada commit realizado por desenvolvedores para o repositório de código remoto, um build será disparado pelo software de integração contínua, de modo que a aplicação seja construída e tenha suas funcionalidades validadas.

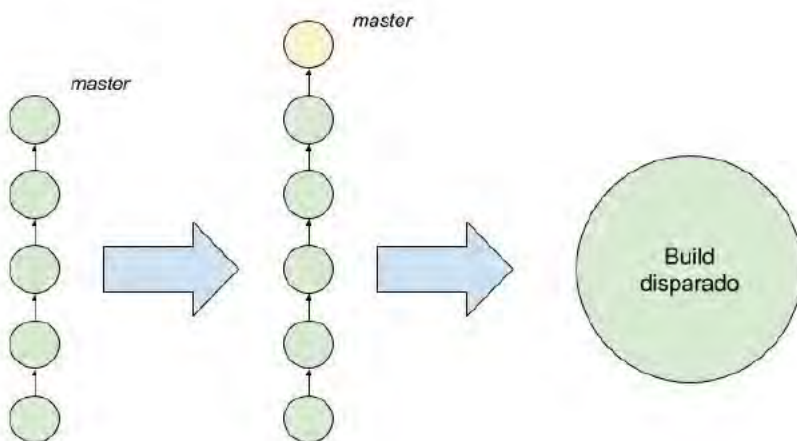


Figura 2.3: Aplicação integrada continuamente



Figura 2.4: Build disparado em software de integração contínua

Dentre os prós referentes a esse modelo de gerenciamento de branches, podem ser citados:

- Simplicidade de manutenção;
- Favorecimento da integração contínua.

Já dentre os contras relacionados a esse modelo de branching

são listados os seguintes:

- Propicia histórico de controle de versão a conter commits que não agregam valor ao software;
- Sobrecarga do software de integração contínua.

Uma vez que todo commit sobre a branch *master* disparará um novo build no software de integração contínua, pode haver um enfileiramento de builds, de modo que, desde que um build esteja sendo executado, todos os outros builds (referentes a commits posteriores) estarão em uma fila de espera aguardando o término da execução do primeiro build:

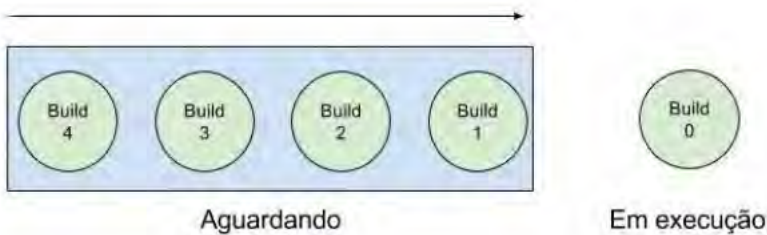


Figura 2.5: Enfileiramento de builds

Para não haver um gargalo na execução dos builds, sob o contexto de gerenciamento de controle de versão através de uma branch, é possível configurar o software de integração contínua para paralelizar a execução dos builds, de forma a reduzir o tempo de espera entre as execuções. *Ok! Mas e se a feature na qual eu estiver trabalhando demandar muito tempo para ser implementada?*

Features dessa categoria, preferencialmente, devem ser associadas a profissionais mais experientes, pois elas terão de ser planejadas, implementadas e integradas gradualmente. Isso para evitar um *integration hell*, que é resultante da integração de muitas modificações sobre o app de uma vez só, em vez do recomendado,

isto é, muitas integrações de pequenas modificações sobre o software.

Além disso, manter versões customizadas de um mesmo app (por exemplo, versões alfa/beta de testes e de produção) torna-se uma tarefa um pouco mais complexa — não é impossível, é somente mais custosa. Pelo fato de o mesmo código estar sendo comitado sobre a mesma branch, deve haver uma forma de comunicar ao software de integração contínua quando determinada alteração sobre uma versão customizada do app deve ser construída de um modo particular. Isso anula a característica de simplicidade desse modelo de branching e, por consequência, faz-se melhor a escolha por outro modelo.

Branching baseado em Gitflow

Gitflow trata-se de um modelo de branching criado por Vincent Driessen, engenheiro e artesão de software holandês. É um modelo multibranch em que cada branch tem seu nome previamente definido pelo modelo, ou segue uma convenção pré-determinada em que cada branch tem uma função bem definida.

Ao iniciar um projeto para desenvolvimento de uma nova aplicação, será criada uma nova branch de nome *develop* a partir da branch *master*. Esta sempre conterá a última versão do software em desenvolvimento.

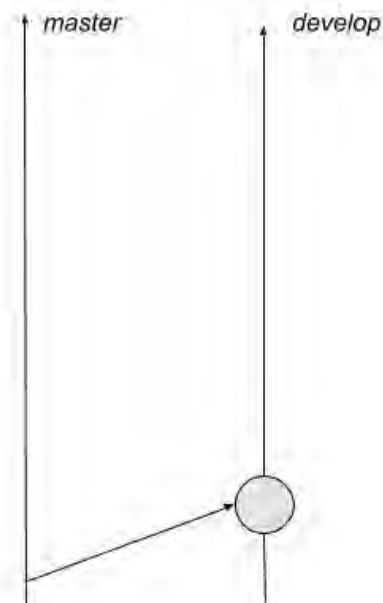


Figura 2.6: Criação da branch develop

Todas as pequenas correções de bugs devem ser comitadas sobre a branch *develop* sem a necessidade de criação de novas branches para isso.

Uma vez que uma nova feature precise ser implementada, essa ação não será realizada na branch *develop*, pois, lembrando, a *develop* somente conterá a última versão comitada do software com features completas ou com pequenas modificações, tais como *bugfixes*. Commits intermediários de novas features não devem ser realizados nessa branch.

Então, uma vez sob esse contexto, deve ser criada uma nova branch partindo da branch *develop* chamada de **feature-branch**. A convenção para nomeação de uma nova *feature-branch* determina que o nome da branch deve ser prefixado com a palavra-chave

feature seguido por um sufixo, como mostrado a seguir:

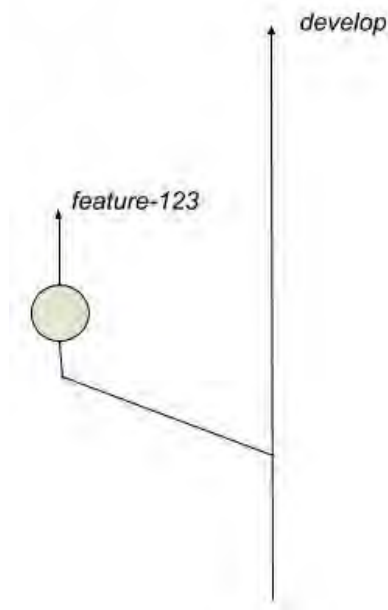


Figura 2.7: Feature branch

No caso da figura anterior, assim que a feature 123 esteja completamente implementada, deve ser realizado o merge da branch *feature-123* em direção à branch *develop*, fazendo com que a branch *develop* tenha a última versão atualizada do software já com a feature 123 implementada dentro de seu próximo pacote de funcionalidades.

Quando for determinado que uma nova versão da aplicação deve ser enviada ao cliente ou publicada no Google Play, por exemplo, uma nova ramificação deve ser criada, de forma que seja possível configurar metadados da aplicação através dela, tais como número da versão a ser publicada e configurações secundárias. Essa nova branch é conhecida como **release-branch**.

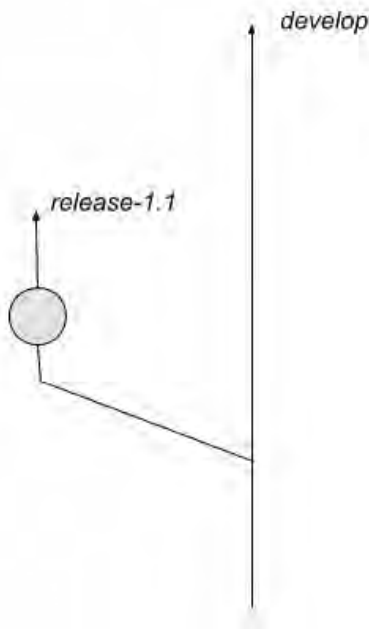


Figura 2.8: Release branch

O nome da *release branch* deve ser prefixado com a palavra-chave *release*, tal como no caso da figura anterior, onde a branch é nomeada como *release-1.1*, e o sufixo é o próximo número de versão da aplicação a ser distribuída. Porém, a nomeação do sufixo é livre.

Assim que todos os tratamentos tenham sido realizados sobre essa branch, deve ser realizado o seu merge em direção à branch *master*, a qual também conterà uma nova tag marcando o nome da versão resultante desse merge. Contudo, a branch de release também deve ser mergeada para a branch *develop*, pois é tido como premissa que a branch *develop* sempre conterà a última versão da aplicação em desenvolvimento. Logo, a *develop* deve conter o número de versão da aplicação atualizado também.

Digamos que exista uma versão já publicada da aplicação no cliente. Então é descoberto um bug crítico nessa aplicação. Qual é a

atitude a ser tomada? Nesse contexto, uma nova branch será criada a partir do último commit realizado sobre a branch *master*, ou seja, a partir da última versão publicada ao cliente. Essa branch é denominada **hotfix-branch**. Com nome prefixado com a palavra-chave *hotfix*, a branch é responsável por conter commits que consertam o bug encontrado em produção:

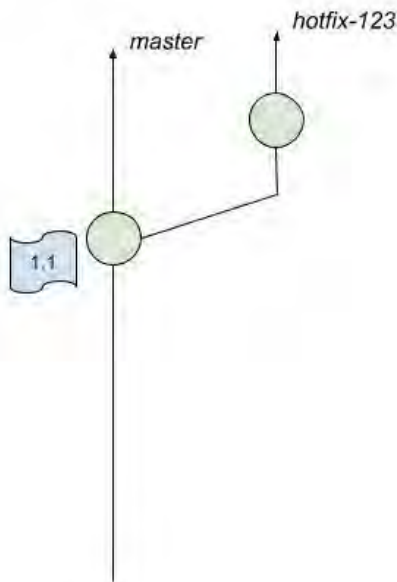


Figura 2.9: Hotfix branch

Tendo sido finalizadas todas as correções sobre o bug, a branch de *hotfix* deve sofrer ação de um merge em direção à branch *master*. Mas não somente. Também deve ser mergeada em direção à branch *develop*. A não ser que exista uma branch *release* viva no sistema de controle de versão, pois, como já mencionado anteriormente, a branch *release* será futuramente mergeada para a branch *develop*.

Somente as branches *master* e *develop* existem durante todo o ciclo de vida da aplicação sob desenvolvimento. As *feature-branches*, *release-branches* e *hotfix-branches*, uma vez que não tenham mais

utilidade, devem ser deletadas.

Ok! Mas afinal, o que isso tem a ver com integração contínua?

Resposta: Tudo! O gerenciamento do ciclo de vida da aplicação é mais bem organizado. Correções são mais fáceis de serem introduzidas. Ou seja, essa manutenção de branches não viabiliza integração contínua. Trata-se de um contraexemplo, pois a frequência com que implementações serão integradas à branch principal da aplicação é indefinida.

Diante desse contexto, o aplicativo somente será integrado em duas situações, costumeiramente:

- Ao ser comitado código sobre a branch *develop*;
- Ao ser aceito um *pull-request* para a branch *master*.

Um **pull-request** é uma solicitação para que sejam integradas modificações sobre um software à branch principal. Simplificando, um *pull-request* é um pedido para um dos responsáveis pelo repositório de código da aplicação, para que se faça um *code review* sobre a implementação realizada. Assim, caso seja aprovado, será iniciado o processo de integração desse código na próxima versão a ser distribuída para o cliente. A nova versão será enviada para o pipeline de integração e, caso validada por seus testes automatizados, será aprovado o merge para a branch *master*.

Commits com incrementos de novas features ou *bugfixes* deverão ser integrados ao app Android mantido sob o controle de versão — seja ele gerenciado sob uma única branch (*master*), ou sob um modelo de branching baseado em Gitflow. E quando nos referimos a uma integração, não somente ela se refere ao ato de concatenar um novo commit a sua branch-alvo, como também a etapa de validação a ser executada sobre esse incremento. Para cada commit, um processo de validação.

Levando em consideração que, por dia, um time de desenvolvimento de tamanho razoável pode comitar uma dezena de vezes, uma dezena de validações seriam realizadas. *Manualmente?* Por quê? Elas podem muito bem ser automatizadas. E é no próximo capítulo que será mostrado como automatizar testes sobre apps Android e através de quais ferramentas.

TESTES AUTOMATIZADOS

As aplicações devem ser testadas. Isso é uma regra. E quando falo "devem ser testadas", não me refiro somente a testes manuais (que também têm a sua importância), mas, principalmente, a testes automatizados. Aplicações não testadas não são confiáveis, encarecem o produto final e são de difícil manutenção. Logo, não há escolha. Simplesmente teste.

E quando falamos sobre testar apps Android, a importância é ainda maior, devido ao vasto ecossistema de dispositivos que existem no mercado. Um aplicativo pode ser aprovado em todos os testes automatizados em um dispositivo de um determinado fabricante, porém reprovado em testes em um dispositivo de outro fabricante. Ou seja, existem testes implementados e o aplicativo ainda falha em um determinado contexto. Imagina se não existissem.

Sob a ótica de viabilizar a entrega contínua de um app Android, testes automatizados são de fundamental importância, pois eles representam alguns dos estágios que formam um pipeline de deployment. Em capítulos anteriores, foi mencionado que o primeiro estágio de um pipeline de deployment é denominado estágio de commit. Apesar de esse estágio conter as subetapas de compilação e empacotamento, neste capítulo, o pipeline de deployment será apresentado sem essas subetapas, de forma que cada estágio do pipeline expresse uma categoria de teste automatizado. Assim, as subetapas de testes unitários e análise

estática de código são desmembradas em dois novos estágios, como mostrado a seguir:



Figura 3.1: Pipeline de deployment automatizado

Em relação ao modelo de pipeline de deployment proposto em capítulos anteriores, esse novo pipeline suprime o estágio de testes manuais, para que o pipeline de deployment seja completamente automatizado. Ou seja, uma vez que um desenvolvedor realize um commit com modificações sobre o aplicativo Android sob desenvolvimento, e todos os testes de todos os estágios sejam executados com correção, ao final do pipeline deve ser publicada uma nova versão do app Android no Google Play automaticamente, sem intervenção humana.

Mas o que significa testes serem automatizados? Quais são seus reais benefícios? Quais tipos de testes existem? Que ferramentas podem ser usadas para executar esses testes contra um aplicativo Android? O aplicativo precisa ser adaptado para poder ser testado? As perguntas são muitas. Seguem suas respostas.

3.1 O QUE É UM TESTE AUTOMATIZADO?

Um **teste automatizado** pode ser executado por um clique de botão ou por um comando em um console, tal que, ao final da execução desse teste, o resultado produzido por ele seja comparado a um resultado esperado. Caso ambos os resultados sejam iguais, pode ser dito que a aplicação **passou** para aquele teste. Porém, caso o resultado produzido seja diferente do esperado, a aplicação **falhou**

para aquele teste.

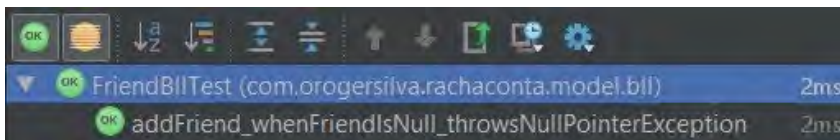


Figura 3.2: Teste executado com sucesso — barra verde — 1

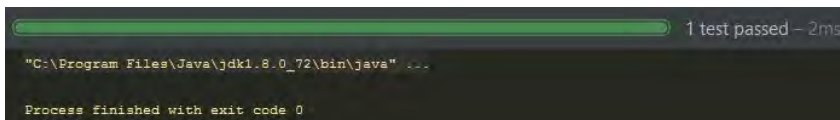


Figura 3.3: Teste executado com sucesso — barra verde — 2

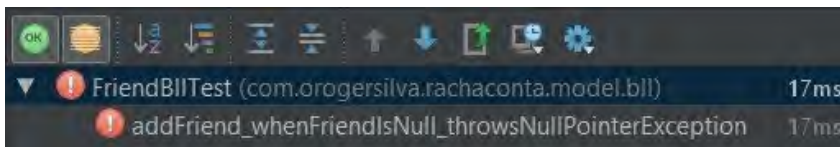


Figura 3.4: Teste executado com falha — barra vermelha — 1

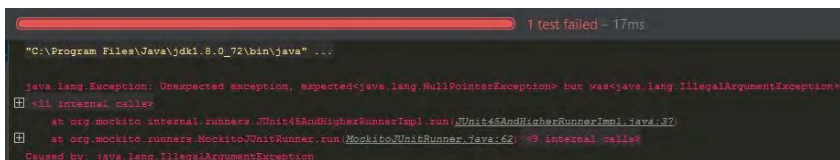


Figura 3.5: Teste executado com falha — barra vermelha — 2

As imagens anteriores mostram duas execuções do mesmo teste contra um aplicativo Android através do Android Studio. Na primeira, o teste passou. Já na segunda, o teste falhou. É interessante notar a barra horizontal em verde e vermelho para o teste executado com sucesso e com falha, respectivamente.

É proposital a barra ocupar boa parte da tela, pois ela facilita a visibilidade sobre o status atual de execução dos testes. Em caso de teste com falha, *stakeholders* serão facilmente notificados sobre algo

estar errado.

Aliás, algumas empresas preferem reforçar esse feedback através de um monitor colocado em uma posição com boa visibilidade no ambiente de trabalho do time de desenvolvimento (e de outras partes interessadas). Ele é dedicado a mostrar, dentre outras informações, resultados de testes automatizados executados contra uma aplicação.



Figura 3.6: Feedback imediato

Uma pergunta pertinente: com a implementação de testes automatizados, testes manuais são ainda necessários? Para responder a essa pergunta, outra ainda precisa ser respondida: "*O que é um teste manual?*". É uma pessoa qualquer interagindo aleatoriamente contra a aplicação-alvo tentando encontrar algum bug?

Na verdade, é completamente o contrário disso. Um teste manual é um procedimento no qual um testador (com *skills* para

descoberta de falhas) interage contra a aplicação sob teste com o auxílio de uma estratégia de testes bem definida, para fins de encontrar bugs e registrá-los para futuro conserto. Quase sempre uma aplicação não consegue ser coberta completamente por testes automatizados, seja por custos, dificuldade em cobrir alguns fluxos de execução ou falta de tempo. Logo, uma vez tendo sido executada uma bateria de testes automatizados, a aplicação de testes manuais faz-se muito útil.

Benefícios

Automatizar testes para um app traz inúmeros benefícios. Alguns merecem destaque.

- **Proteção contra a introdução de novos bugs**

Esse é o benefício principal. Para exemplificar, veja uma suíte com métodos de teste.

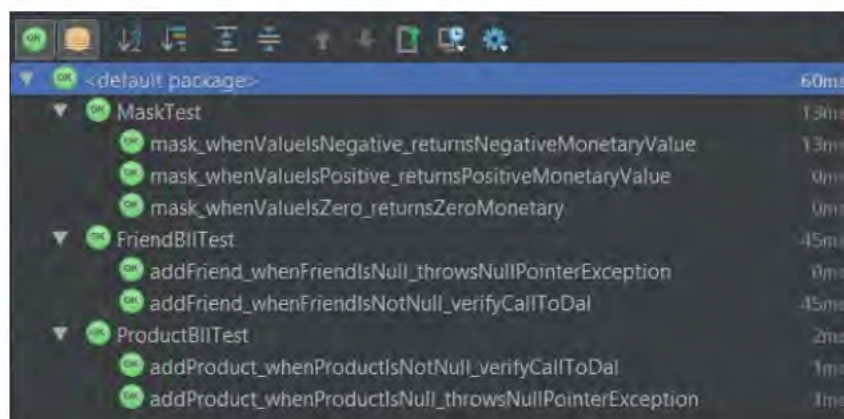


Figura 3.7: Suíte de métodos de teste

A imagem mostra uma aplicação sendo aprovada para um conjunto de métodos de teste. Porém, em um cenário hipotético, a empresa proprietária dessa aplicação contratou um novo

desenvolvedor. Então, esse desenvolvedor olha para o seguinte método do app:

```
public void addFriend(Friend friend) {  
    if (friend == null) {  
        throw new NullPointerException();  
    }  
  
    mFriendDal.create(friend);  
}
```

Ele pensa em voz alta: *Hum, do meu ponto de vista, não parece correto esse método lançar um `NullPointerException`. Vou dar um jeito nisso.*

```
public void addFriend(Friend friend) {  
    if (friend == null) return;  
  
    mFriendDal.create(friend);  
}
```

Bom, parece que o novo desenvolvedor está motivado e já quis mostrar suas competências. Entretanto, seu colega de trabalho atualizou o código-fonte da aplicação em sua máquina já com as modificações do novo desenvolvedor. Então, ele resolve executar os testes automatizados para verificar se algo está errado e...

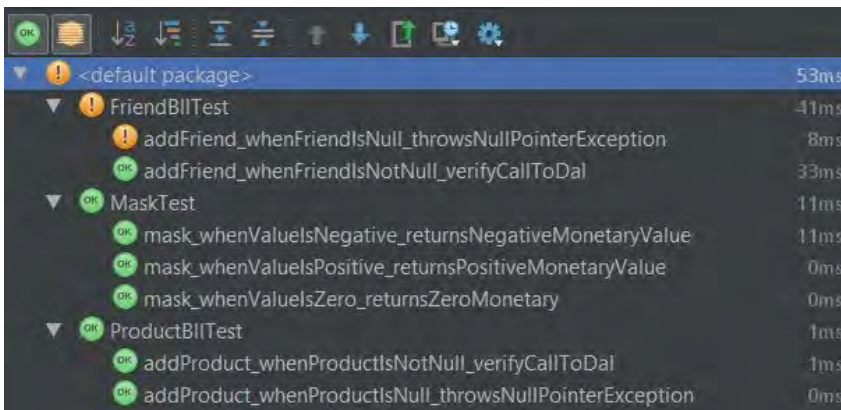


Figura 3.8: Suíte falhando testes

```
"C:\Program Files\Java\jdk1.7.0_79\bin\java" ...

java.lang.AssertionError: Expected exception: java.lang.NullPointerException
[+] testWhenNullThrows()
    at org.mockito.internal.runners.DefaultInternalRunner$1.runTest(DefaultInternalRunner.java:47)
    at org.mockito.runners.MockitoJUnitRunner.run(MockitoJUnitRunner.java:62)
    at com.intellij.rt.execution.junit.JUnit4TestRunner.executeMain(JUnit4TestRunner.java:62)

Process finished with exit code -1
```

Figura 3.9: Falha no console da suíte de testes

Oops! Parece haver algo errado. Um teste está falhando. Reparem o nome do teste que falha:



Figura 3.10: Método de teste falhando

Testes devem ter bons nomes, pois facilita a investigação do motivo do aparecimento de um novo problema na aplicação. O nome do método de teste deixa claro: ao executar o método `addFriend`, caso o usuário seja nulo, deve ser lançado um objeto `NullPointerException`. Que não é o caso, pois o novo desenvolvedor modificou o código para `addFriend` não retornar qualquer valor quando a condição for atendida.

Quando um teste falha, significa que a aplicação não está aprovada para o teste aplicado. Isso não significa que a aplicação está completamente quebrada. Tudo depende do impacto causado em outros métodos que fazem uso do método `addFriend`. Mas uma coisa é certa: o método de teste falhado não pode permanecer assim. Uma aplicação deve ser aprovada em 100% dos seus testes (99,9% não é 100%, que isso fique bem claro).

- **Facilitação de refatorações**

Refatorar é o exercício mais eficiente para ser um desenvolvedor melhor dia após dia.

Refatorar não é modificar o código-fonte de qualquer forma. Refatorar é tornar menos custosa futuras manutenções em uma aplicação, e tornar uma aplicação escalável.

Então, se refatorar com frequência é necessário, como ter a confiança de que alterações no código-fonte não introduzirão bugs na aplicação? Através de testes automatizados, como já foi mencionado anteriormente.

- **Auxílio na definição da arquitetura**

A arquitetura de uma aplicação, em grande parte das vezes, não deve ser completamente definida antes de sua codificação.

Esse é um benefício que pouco é difundido. Conforme novos testes automatizados são implementados, no início da codificação de uma aplicação, possíveis equívocos arquiteturais podem ser encontrados. Através da prática de *Test Driven Development* (TDD) — em que testes automatizados são escritos antes das funcionalidades que eles devem cobrir —, o desenvolvedor tem a chance de enxergar possíveis gaps de arquitetura em sua aplicação o mais cedo possível, pois os testes descrevem cenários que a aplicação sob teste deve atender.

Logo, um contato prévio com esses cenários, através da implementação dos testes, traz uma visão mais ampla, como

também o modo como está evoluindo a aplicação arquiteturalmente. Assim, boa parte de reescrita da aplicação pode ser evitada caso gaps de arquitetura sejam detectados antecipadamente pela implementação de testes automatizados.

3.2 TESTES UNITÁRIOS

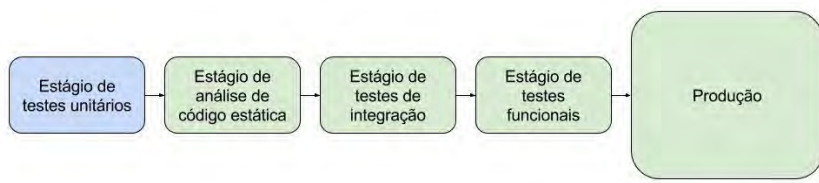


Figura 3.11: Estágio de testes unitários

Testes unitários testam a unidade. Bem, mas o que é uma unidade? Resumindo, uma unidade pode ser uma `Activity`, um `Fragment`, um componente de acesso à camada de dados, um componente de acesso à rede etc. Ou seja, são os menores componentes testáveis de uma aplicação.

Cada método de teste unitário deve atuar *somente* sobre uma unidade de cada vez.

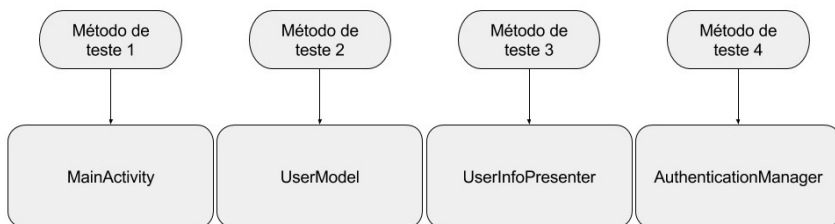


Figura 3.12: Métodos de teste unitários

Caso uma unidade dependa de outra, como testá-las separadamente? É aí que entra em cena os objetos mock. O **mock**

simulará a unidade que a unidade dependente faz uso, o que viabiliza a aplicação do teste unitário.

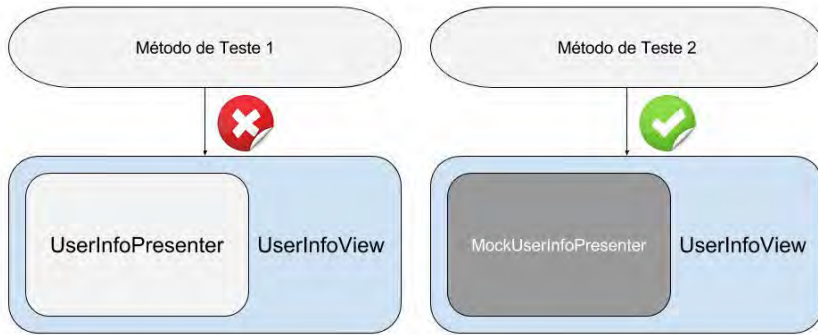


Figura 3.13: Mock em testes unitários

Uma vez apresentada a parte conceitual, vamos a um exemplo prático.

Mãos à obra

A seguir, serão abordadas estruturas do app *Racha Conta*, já mencionado anteriormente, para exemplificar a implementação de testes unitários. Serão implementados testes unitários para duas classes: `FriendBll` e `DeskActivity`.

A primeira classe pertence à camada de lógica de negócio e contém operações CRUD (*Create-Retrieve-Update-Delete*) sobre objetos que representam usuários. A segunda classe contém a lógica de gerenciamento da tela de lista de usuários.

Começando pela classe `FriendBll`. Ela contém um método chamado `getFriend`:

```
public Friend getFriend(String name) {  
  
    Friend friend = null;  
  
    try {
```

```

        if (StringUtils.isNullOrEmpty(name)) {
            throw new InvalidStringException();
        }

        friend = mFriendDal.retrieve(name);

    } catch (InvalidStringException e) {}

    return friend;
}

```

Esse método retorna um objeto do tipo `Friend` com todas as informações sobre um usuário, uma vez fornecido o seu nome. Esse método pode ser testado de diversas formas.

Por exemplo, caso o nome passado para `getFriend` seja nulo, qual é o resultado esperado a ser produzido por esse método? De acordo com o fluxo de execução, o nome é avaliado na expressão condicional e é disparado um `InvalidStringException`. O fluxo de execução segue pelo bloco `catch` e, por fim, atinge o comando `return`. O valor do objeto `friend` nesse ponto será nulo, pois, desde que foi inicializado com esse valor, ele não foi alterado uma vez sequer. Assim, um método de teste correspondente a esse fluxo será definido.

O *source set* referente aos testes unitários no projeto da aplicação é identificado por (*test*).

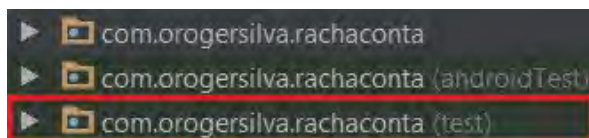


Figura 3.14: Source set para testes unitários

Nesse *source set* é criada a classe `FriendBllTest`, que será a classe responsável por conter testes unitários referente à unidade `FriendBll`.

O método de teste que testa o fluxo mencionado anteriormente é:

```
@Test
public void getFriend_whenFriendNameIsNull_returnsNull() {

    // ARRANGE
    final String NULL_FRIEND_NAME = null;

    // ACT
    Friend gottenNullFriend = mFriendBll.getFriend(NULL_FRIEND_N
AME);

    // ASSERT
    assertNull(gottenNullFriend);
}
```

Todo método de teste unitário deve ser precedido por um *annotation* `@Test`, de modo que o método seja reconhecido como um teste.

Um bom padrão para nomeação de métodos de teste unitários pode ser descrito como:

[nome do método testado]_[condição]_[valor produzido pelo método]

Métodos devem ter bons nomes. Alguém poderia perguntar: *Mas o nome do método não está muito longo?* Preferencialmente, nomes de métodos devem ser curtos. Porém, o principal é o método expressar sua intenção. Se é difícil encontrar um nome enxuto que expresse a intenção do método, não hesite: nomeie-o com um nome longo. Seus colegas desenvolvedores (e você mesmo) agradecerão futuramente.

O corpo do método segue um *pattern* denominado **Arrange-Act-Assert** (AAA). Ele auxilia a organização do teste. A seção *Arrange* contém todas as instruções necessárias para preparar a invocação e definição de valor esperado a ser retornado pelo método sendo testado.

```
// ARRANGE
```

```
final String NULL_FRIEND_NAME = null;
```

A seção *Act* contém a invocação do método sendo testado. O valor retornado pelo método deve ser armazenado para que esse resultado seja verificado.

```
// ACT
Friend gottenNullFriend = mFriendBll.getFriend(NULL_FRIEND_NAME);
```

A seção *Assert* contém todo código-fonte responsável por comparar o valor retornado pelo método sob teste com um resultado esperado, que, no caso desse teste, é nulo.

```
// ASSERT
assertNull(gottenNullFriend);
```

Para executar o teste unitário, pressione o ícone assinalado na figura a seguir:



Figura 3.15: Executando método de teste

E como resultado do teste...

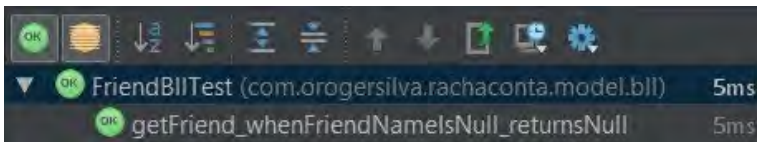


Figura 3.16: Resultado da execução do método de teste — 1

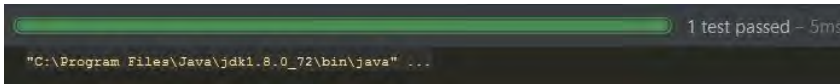


Figura 3.17: Resultado da execução do método de teste — 2

O resultado do teste com sucesso significa que, para o fluxo coberto do método testado, ele foi realizado com sucesso. Mas somente para aquele fluxo. Logo, outros fluxos do método `getFriend` precisam ser cobertos.

Implementando outro método de teste para cobrir outro fluxo de `getFriend` :

```
@Test
public void getFriend_whenFriendNameIsValid_returnsFriend() {

    // ARRANGE
    final String FAILED_TEST_MESSAGE = "Objeto 'Friend' recupera
do deve ser igual ao objeto 'Friend' esperado.";

    final String VALID_FRIEND_NAME = "Roger";
    final double FRIEND_DEBT = 34.00;

    Friend expectedFriend = new Friend(VALID_FRIEND_NAME, FRIEND
_DEBT);

    when(mFriendDalMock.retrieve(VALID_FRIEND_NAME)).thenReturn(
expectedFriend);

    // ACT
    Friend gottenValidFriend = mFriendBll.getFriend(VALID_FRIEND
_NAME);

    // ASSERT
    assertEquals(FAILED_TEST_MESSAGE, expectedFriend, gottenVali
dFriend);
}
```

Esse método de teste trata do fluxo (como é dito em seu nome) do cenário em que, quando passado como parâmetro um nome de usuário válido a `getFriend`, será retornado um objeto `Friend` com informações completas sobre um usuário. Analisando o fluxo

de execução de `getFriend`, no contexto sendo tratado, pode ser visto que é realizada uma invocação ao método `retrieve` em um objeto da classe `FriendDal`. Ou seja, deverá ser usado um objeto *mock* de `FriendDal` para simular uma invocação a `retrieve`, para que esse fluxo possa ser testado de forma unitária.

A classe `FriendBll` tem um construtor que recebe como argumento um objeto `FriendDal`. Logo, `FriendBll` será instanciado recebendo como argumento um objeto *mock* de `FriendDal`. Isso permitirá à unidade `FriendBll` ser testada de forma isolada. A instanciação é realizada desta forma:

```
@Before
public void setup() {

    mFriendBll = new FriendBll(mFriendDalMock);
}
```

O método `setup` é precedido pela *annotation* `@Before`. Esta determina que `setup` será invocado antes da execução de cada método de teste definido em `FriendBllTest`. Esse tratamento é realizado para que todas as pré-condições sejam atendidas para a execução de cada método de teste a ser executado. O objeto *mock* é declarado precedendo o objeto-alvo com a *annotation* `@Mock`:

```
@Mock
FriendDal mFriendDalMock;
```

O método de teste contém o seguinte comando na seção *Arrange*:

```
when(mFriendDalMock).retrieve(VALID_FRIEND_NAME)).thenReturn(expectedFriend);
```

Esse comando deve ser interpretado da seguinte forma: quando o fluxo de execução se deparar com uma chamada ao método `retrieve` de um objeto do tipo `FriendDal`, o valor retornado por esse método deve ser `expectedFriend`. O método não será invocado. Será realizado um "faz de conta". Isso permite com que

somente `FriendBll` esteja sendo testado pelo método de teste.

Um outro estilo de teste unitário pode ser implementado para a unidade `DeskActivity`. Criamos a classe `DeskActivityTest` para conter métodos de teste para testar a classe `DeskActivity`. Como `DeskActivity` gerencia a lógica de uma das telas do app, é dito que os testes implementados para essa classe são **testes unitários de view**.

Como eles são testes que exigem manipulações de objetos de classes presentes no Android SDK (*Software Development Kit*), será necessário um `Robolectric`. Veja a seguir:

```
@SmallTest
@RunWith(RobolectricGradleTestRunner.class)
@Config(constants = BuildConfig.class, sdk = Build.VERSION_CODES
.LOLLIPOP)
public class DeskActivityTest {
```

O fluxo de execução que será tratado é expresso pelo método de teste:

```
@Test
public void clickingOnFloatingActionButton_shouldStartDialog() {

    // ARRANGE
    FloatingActionButton addFriendFloatingActionButton = (FloatingActionButton) mDeskActivity.findViewById(R.id.desk_floatingactionbutton);

    InputAddFriendDialog inputAddFriendDialog;

    // ACT
    addFriendFloatingActionButton.performClick();

    inputAddFriendDialog = (InputAddFriendDialog) ShadowDialog.getLatestDialog();

    // ASSERT
    assertTrue(inputAddFriendDialog.isShowing);
}
```

Ou seja, pressionado o botão para a adição de um novo usuário

na lista de usuários, a lógica de execução deve iniciar um diálogo, ou seja, uma janela para o preenchimento do nome desse novo usuário. O diálogo não será realmente iniciado. Robolectric somente verifica se, caso ao pressionar o botão para a adição de um novo usuário, o fluxo de execução dispara uma chamada para a exibição do diálogo `InputAddFriendDialog`, sem realmente exibi-lo de fato.

Lembrando, não é necessário um emulador ou dispositivo físico para executar esse teste. O teste é iniciado na JVM (*Java Virtual Machine*). Isso viabiliza a eficiência do uso de TDD, por exemplo, pois, como testes são executados com frequência durante o período de desenvolvimento, é primordial a rapidez da sua execução.

3.3 ANÁLISE ESTÁTICA DE CÓDIGO

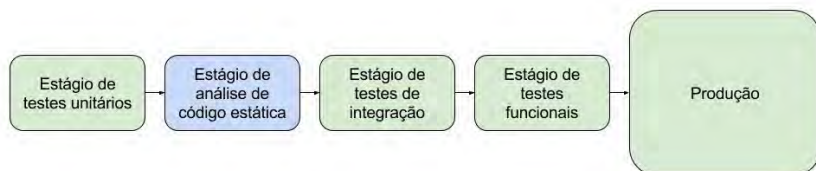


Figura 3.18: Estágio de análise estática de código

Existe outra categoria de testes que é responsável por: validar se o estilo de escrita do código-fonte condiz com o esperado; apontar *bad smells* no código que sejam indícios de ocorrências de bugs no aplicativo quando em execução; e sinalizar outras medições de qualidade de código, tais como performance, usabilidade, corretude, dentre outras. A categoria de testes automatizados por tratar dessas questões é denominada **análise estática de código**.

A adoção do uso por tal categoria de testes deve ser mensurada. Em projetos de baixa escala, com poucos membros no time de desenvolvimento, talvez ela não se justifique, pois testes como esses

têm pouco ROI (*Return On Investment*) sob esse contexto. Vale mais a pena dedicar o esforço à implementação de testes unitários e de integração, por exemplo.

Já em times com muitos membros, faz-se mais adequada a adoção de análise estática de código em projetos. Por haver pluralidade de estilo de escrita de código em grandes times, um processo automatizado que trata de verificações sobre a saúde do código-fonte não é somente recomendada, mas sim necessária.

Há uma boa variedade de ferramentas que realiza esse tipo de tratamento. Elas serão detalhadas a seguir. Porém, para fins de organização, será adotada uma convenção para manter todas as configurações responsáveis por tratar da análise de código.

As configurações responsáveis por tratar a análise de código podem ser mantidas de diversas formas. Agora, mantê-las de forma organizada tornando-as de fácil manutenção é sempre preferível. Logo, será criado um folder `config` na raiz do projeto da aplicação Android, onde todas as configurações sobre qualidade de código serão mantidas.

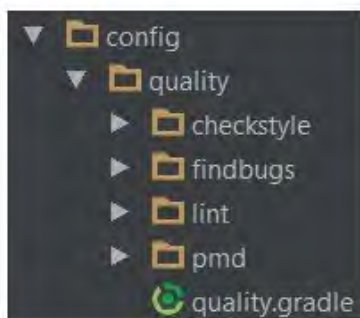


Figura 3.19: Organização de folders para análise estática de código

Um subfolder `quality` será também criado para manter somente as configurações de qualidade de código. E dentro desse

último folder, são mantidas as configurações de cada ferramenta de análise de código adotada, assim como um arquivo Gradle que conterá a descrição de cada *task* responsável por tratar cada tipo de análise.

Além disso, deve ser adicionada no arquivo `build.gradle`, em nível de projeto, a aplicação de todas as *tasks* definidas em `quality.gradle`:

```
apply from: "${rootProject.getRootDir()}/config/quality/quality.gradle"
```

Figura 3.20: Aplicação das configurações para análise de código

Checkstyle

Trata-se de uma ferramenta que auxilia desenvolvedores a manterem um padrão de escrita do código Java em seus projetos.

O sistema de build Gradle já contém a ferramenta Checkstyle. Assim, basta ativá-la para que seja executada durante o build. Para isso, no arquivo `quality.gradle`, aplique o plugin de Checkstyle.

```
Aplicação do plugin Checkstyle — apply plugin:  
'checkstyle'
```

Também, deve ser definida a *task* a ser executada quando for lançado o Checkstyle, como mostrado a seguir:


```

task checkstyle(type: Checkstyle) {
    configFile file("${project.rootDir}/config/quality/checkstyle/checkstyle.xml")
    .checkstyleSuppressionsPath =
        file("${project.rootDir}/config/quality/checkstyle/suppressions.xml").absolutePath
    source 'src'
    include '**/*.java'
    exclude '**/gen/**'
    classpath = files()
}

```

Figura 3.21: Task do Checkstyle

Para realizar a verificação com o Checkstyle, execute o seguinte comando Gradle:

Gradle Windows — `gradlew.bat checkstyle`

Gradle Linux — `./gradlew checkstyle`

Para exemplificar, digamos que o seguinte tipo genérico seja descrito da seguinte forma:

```
private List< Friend > mFriends;
```

Ou seja, há espaços em branco na declaração desse tipo. Após a verificação do `checkstyle`, é gerado um arquivo HTML contendo o resultado da análise. No caso desse exemplo, é sinalizada a má prática de escrita de um tipo genérico com espaços em branco.

Error Description

'<' é seguido de espaço em branco.

'>' é precedido por espaço em branco.

Figura 3.22: Report do Checkstyle

Findbugs

É uma ferramenta de análise estática que busca por *bug patterns* em *bytecodes* Java. Ou seja, sobre arquivos `.class`, além de indicar potenciais problemas de segurança.

Assim como Checkstyle, Findbugs já é incluído no sistema de build Gradle. Assim, deve-se aplicar seu plugin no `quality.gradle`:

Aplicação do plugin Findbugs — `apply plugin: 'findbugs'`

Também deve ser definida a *task* responsável pelo modo como deve ser configurado o Findbugs.

```
task findbugs(type: FindBugs, dependsOn: assembleDebug) {  
  
    ignoreFailures = false  
    effort = "max"  
    reportLevel = "high"  
    excludeFilter =  
        new File("${project.rootDir}/config/quality/findbugs/findbugs-filter.xml")  
    classes = files("${project.rootDir}/app/build/intermediates/classes")  
  
    source 'src'  
    include '**/*.java'  
    exclude '**/gen/**'  
  
    reports {  
  
        xml.enabled = false  
        html.enabled = true  
  
        xml {  
            destination "$project.buildDir/reports/findbugs/findbugs.xml"  
        }  
  
        html {  
            destination "$project.buildDir/reports/findbugs/findbugs.html"  
        }  
    }  
}
```

Figura 3.23: Task do Findbugs

Para o uso dessa ferramenta, deve ser executada a seguinte task do Gradle:

```
Gradle Windows — gradlew.bat findbugs
```

```
Gradle Linux — ./gradlew findbugs
```

PMD

Semelhante à ferramenta Findbugs, porém analisa o código-fonte da aplicação, e não somente os *bytecodes*. Além disso, PMD não somente é aplicável à linguagem Java, como também para outras linguagens de programação.

Tal como as outras ferramentas já apresentadas, PMD é incluído no sistema de build Gradle por padrão. Basta a aplicação de seu plugin em `quality.gradle` :

```
Aplicação do plugin PMD — apply plugin: 'pmd'
```

Também, sua *task* dedicada deve ser declarada no mesmo arquivo.

```

task pmd(type: Pmd) {

    ignoreFailures = false
    ruleSetFiles =
    |   files("${project.rootDir}/config/quality/pmd/pmd-ruleset.xml")
    ruleSets = []

    source 'src'
    include '**/*.java'
    exclude '**/gen/**'

    reports {

        xml.enabled = false
        html.enabled = true

        xml {
            destination "${project.buildDir}/reports/pmd/pmd.xml"
        }

        html {
            destination "${project.buildDir}/reports/pmd/pmd.html"
        }
    }
}

```

Figura 3.24: Task do PMD

Para o acionamento da ferramenta, deve ser executado através do console o seguinte comando Gradle:

Gradle Windows — `gradlew.bat pmd`

Gradle Linux — `./gradlew pmd`

Com PMD, é possível realizar uma análise mais criteriosa sobre o código-fonte em relação à ferramenta Findbugs. Por exemplo, o conhecido dilema do "abre-fecha chaves" em instruções condicionais pode ser definido em sua configuração.

Lint

Lint é uma ferramenta de análise estática de código voltada a soluções Android, que realiza validações tais como correção, usabilidade, segurança, dentre outros atributos e, também, em aspectos exclusivos de projetos Android, tais como a obrigatoriedade da disposição de figuras de dimensões distintas em seus respectivos folders `drawable`.

A seguir, veja as configurações necessárias para a implantação de Lint no arquivo `quality.gradle`:

```
android {  
    lintOptions {  
        abortOnError true  
        disableWarnings false  
        htmlReport true  
        lintConfig file("${project.rootDir}/config/quality/lint/lint.xml")  
        htmlOutput file("${project.buildDir}/reports/lint/lint-result.html")  
        xmlOutput file("${project.buildDir}/reports/lint/lint-result.xml")  
    }  
}
```

Figura 3.25: Configurações para o Lint

Para analisar o código-fonte, deve ser disparada a *task* relacionada à ferramenta Lint na execução do build Gradle.

Gradle Windows — `gradlew.bat lint`

Gradle Linux — `./gradlew lint`

Uma pergunta pertinente: *Mas por que tantas ferramentas se elas servem para resolver o mesmo problema?* Bom, algumas delas

resolvem o mesmo problema. Porém, elas o fazem de diferentes formas. Ou seja, elas se complementam. A existência de uma ferramenta não exclui a utilidade da outra. Logo, todas podem ser usadas em conjunto de forma a analisar o código-fonte de um aplicativo Android por completo.

Se for requerido executar todas essas ferramentas durante o mesmo build, pode ser declarada uma dependência delas com a task `check` do Gradle, de forma com que todas sejam disparadas de forma implícita.

```
check.dependsOn 'checkstyle', 'findbugs', 'pmd'
```

Figura 3.26: Dependência de tasks para análise de código

Por padrão, a ferramenta Lint é acionada ao disparar a task `check`.

Outro ponto a ser frisado são customizações sobre quais regras de convenção de código devem ser aplicadas por cada ferramenta. As ferramentas permitem a descrição dessas customizações em arquivos dedicados, como mostrado a seguir:

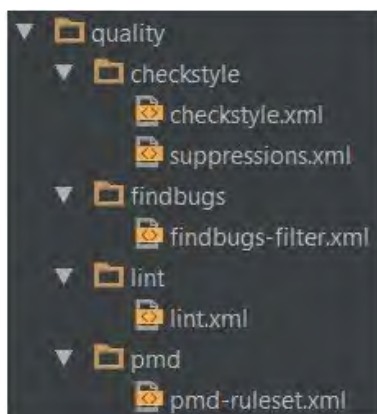


Figura 3.27: Customizações para análise de código

3.4 TESTES DE INTEGRAÇÃO

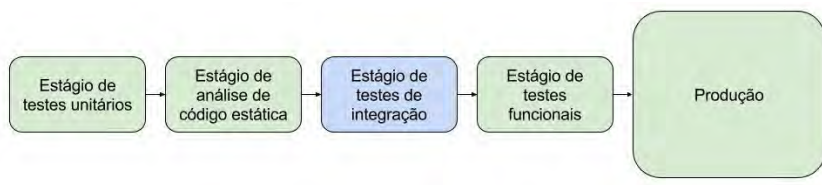


Figura 3.28: Estágio de testes de integração

Dado que as unidades de uma aplicação são validadas separadamente pelos seus correspondentes testes unitários, nada garante que a aplicação esteja livre de bugs. Afinal, as unidades precisam interagir umas com as outras, de modo que a aplicação solucione os problemas do usuário. Logo, as unidades e interações entre elas também devem ser testadas:

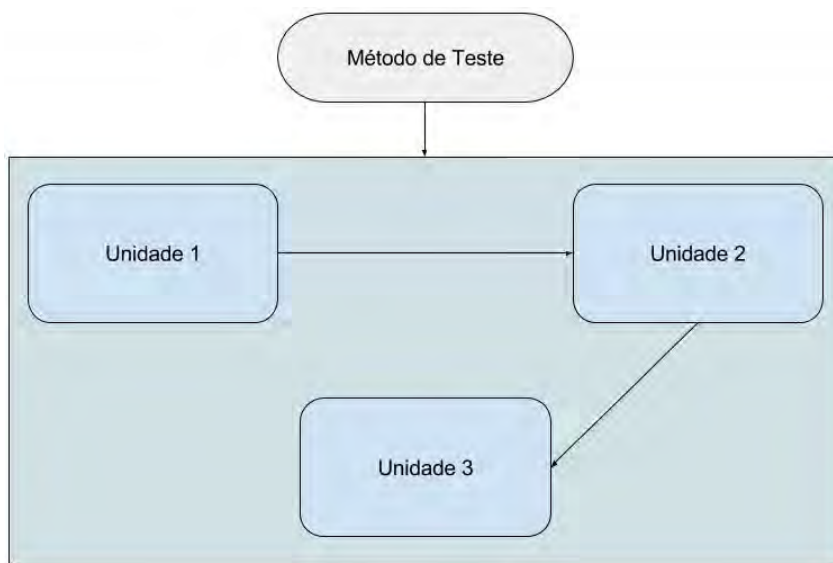


Figura 3.29: Teste de integração

Um **teste de integração** é responsável por validar as interações entre componentes da aplicação em si ou também entre

componentes da aplicação e partes do sistema com o qual ela interage.

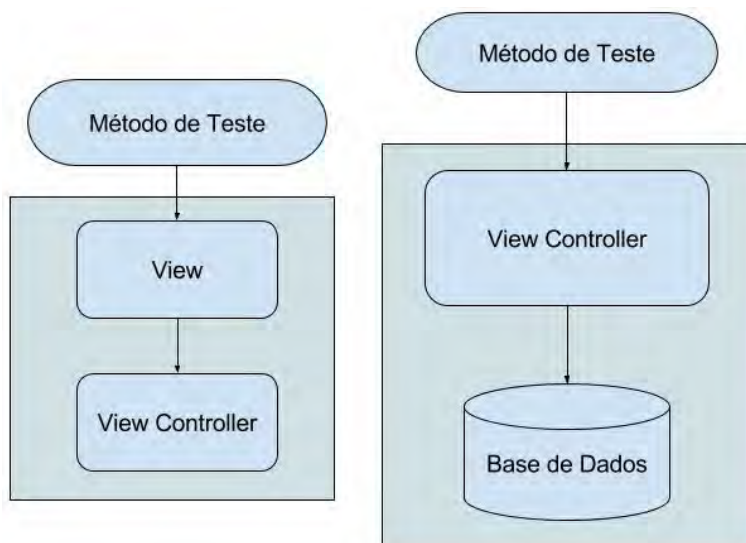


Figura 3.30: Testando interações entre componentes da aplicação e componentes externos

Testes de integração estão aí para atender necessidades, tais como as listadas a seguir.

Mais abrangentes que testes unitários e com maior granularidade que testes funcionais

Testes de integração agem sobre a interação entre componentes. Em testes unitários, as unidades não interagem. Logo, existe essa demanda a ser validada por testes de integração.

Alguém pode perguntar: *Mas testes funcionais podem tratar disso também.* Podem mesmo. Mas vejamos o seguinte trecho de código. Ele mostra um método pertencente à camada de acesso a dados (logo, não coberto por testes unitários, já que essa camada realiza interações diretamente com uma base de dados):


```

public int create(Friend friend) {

    mRealm.beginTransaction();

    try {

        mRealm.copyToRealm(friend);

    } catch (IllegalArgumentException | RealmPrimaryKeyConstraintException e) {

        mRealm.cancelTransaction();
        throw e;
    }

    mRealm.commitTransaction();

    return SUCCESS_OPERATION;
}

```

A pergunta é: como construir o cenário de teste que cubra o fluxo que faz com que seja lançada uma exceção `IllegalArgumentException`? Vale lembrar que testes funcionais são baseados em BDD (*Behavior Driven Development*). Logo, por que um cenário tão granular deveria fazer parte de um grupo de testes responsável por cobrir cenários referentes a regras de negócio? Os testes de integração estarão aí para atender esse problema.

Executam mais rapidamente que testes funcionais

Apesar de serem mais lentos do que testes unitários, testes de integração executam mais rapidamente do que testes funcionais. Logo, ainda que sua execução não seja adequada quando usada a abordagem TDD (por não ser tão rápida quanto a de testes unitários), os testes precisarão ser executados com uma certa frequência ainda.

Portanto, se testes onde componentes não podem ser mockados necessitam ser executados com uma certa frequência, que seja uma

bateria de testes de integração em vez de testes funcionais que, apesar de sua utilidade, pecam pelo longo tempo de execução.

Aumentam a cobertura de código testado

Testes unitários testam unidades separadamente. Logo, com a atuação de testes de integração sobre a interação entre diferentes componentes, fluxos de execução da aplicação que antes não eram cobertos passarão a ser. Isso aumenta o grau de cobertura de código da aplicação.

Testam interações entre componentes dependentes do dispositivo

Quando é necessário o uso de alguma funcionalidade nativa da API do Android, não é incomum o aparecimento de bugs causados por alguma versão específica da API. Por exemplo, uma chamada de método que passou a ser *deprecated* após um período de tempo.

Dependências de dispositivo também são um problema frequente a ser tratado. Assim, sabendo da existência de tais dependências, é recomendável que elas sejam testadas o mais previamente possível. Testes de integração podem suprir essa demanda. Afinal, por que deixar os bugs pipocarem no dispositivo do usuário, quando eles poderiam ter sido evitados em tempo de desenvolvimento por testes de integração?

3.5 TESTES DE INTEGRAÇÃO EM ANDROID

O Android Studio faz uma separação dos diferentes grupos de testes para aplicativos da plataforma Android. E essa separação pode ser feita da seguinte forma:

- **Unit Tests:** são executados na JVM, não acessam a API

- **Instrumentation Tests:** acessam a API real de Android, logo, precisam que um emulador esteja disponível para a execução de seus testes.

- **Instrumentation Tests:** acessam a API real de Android, logo, precisam que um emulador esteja disponível para a execução de seus testes.

Assim, testes de integração, no contexto de desenvolvimento Android, são definidos como *instrumentation tests* (testes de instrumentação). E para o tratamento desses tipos de testes em apps Android, um requisito obrigatório é o uso do framework *JUnit*. Outros frameworks podem ser adotados também, como os que facilitam os testes envolvendo a camada de acesso à rede.

Sugiro a leitura de um excelente post, escrito por Matt Swanson (2014), que trata sobre testes de integração envolvendo elementos de camada de acesso à rede.

Por motivos de simplicidade, vou tratar sobre testes de integração envolvendo somente a camada de acesso a dados, com o objetivo de facilitar a compreensão da essência de tais testes.

Mãos à obra

O primeiro método de teste de integração atuará sobre o método `create` da classe `FriendDal`, ou seja, de uma classe da camada de acesso a dados. A classe de teste `FriendDalTest`, formada por *instrumentation tests*, é definida da seguinte forma:

```
@MediumTest
@RunWith(AndroidJUnit4.class)
public class FriendDalTest {
```

O método de teste

`create_whenProductIsNull_returnsIllegalArgumentException` (como o próprio nome descreve) trata do cenário de um objeto `Friend` nulo passado como parâmetro para o método testado, resultando no disparo de um `IllegalArgumentException` :

```
@Test(expected = IllegalArgumentException.class)
public void create_whenFriendIsNull_returnsIllegalArgumentException() {

    // ARRANGE
    Friend nullFriend = null;

    // ACT
    mFriendDal.create(nullFriend);
}
```

É interessante notar como o objeto `mFriendDal` é instanciado:

```
@BeforeClass
public static void setupClass() {

    mContext = InstrumentationRegistry.getTargetContext().getApplicati
onContext();
    mFriend = new FriendDal(mContext);

    mFriendDal.clearDatabase();
}
```

Ou seja, diferente do cenário tratado na seção sobre testes unitários, o objeto da classe `mFriendDal` é instanciado de verdade, e não um *mock* desse objeto. Para isso, é obtido o contexto de execução dos testes através de `InstrumentationRegistry.getTargetContext().getApplicati onContext()` , para que seja passado como parâmetro ao construtor de `FriendDal` .

O contexto é necessário para a instanciação do objeto que acessa a base de dados. Nessa aplicação, fiz uso da biblioteca Realm (<https://realm.io/>) para isso. Trata-se de um excelente framework para manipulação de bases de dados em aplicações mobile. Porém, qualquer outro framework para manipulação da base de dados poderia ter sido usado.

Um segundo método de teste que merece atenção é mostrado a seguir:

```
@Test
public void retrieve_whenFriendNameExists_returnsFriend() {

    // ARRANGE
    final String FRIEND_NAME = "Roger";
    final double FRIEND_DEBT = 52.00;

    final double DEBT_DELTA = 0.01;

    Friend friend = new Friend(FRIEND_NAME, FRIEND_DEBT);

    Realm realm = Realm.getInstance(mContext);

    realm.beginTransaction();
    realm.copyToRealm(friend);
    realm.commitTransaction();

    // ACT
    Friend retrievedFriend = mFriendDal.retrieve(FRIEND_NAME);

    // ASSERT
    assertEquals(friend.getName(), retrievedFriend.getName());
    assertEquals(friend.getDebt(), retrievedFriend.getDebt(), DEBT_DELTA);
}
```

Ele cobre o cenário em que, já existindo um usuário buscado na base de dados, o método `retrieve` retorna o objeto `Friend` referente a esse usuário. Porém, o detalhe mais interessante sobre

esse método é a estratégia de teste. Como o método testado é `retrieve`, pertencente à classe `FriendDal`, é recomendável que nenhum outro método da classe `FriendDal` seja mencionado nesse método de teste.

Todos os métodos presentes nessa classe serão julgados por uma bateria de testes. Caso o método para criação de usuário (`create`) seja usado para criar o usuário no método de teste `retrieve_whenFriendNameExists_returnsFriend`, uma falha em `create` pode comprometer totalmente sua credibilidade. Assim, deve ser usada outra abordagem para a criação do usuário na base de dados, para que, por fim, o método `retrieve` seja corretamente testado. Essa preparação pode ser notada na seção *Arrange*.

```
// ARRANGE
final String FRIEND_NAME = "Roger";
final double FRIEND_DEBT = 52.00;

final double DEBT_DELTA = 0.01;

Friend friend = new Friend(FRIEND_NAME, FRIEND_DEBT);

Realm realm = Realm.getInstance(mContext);

realm.beginTransaction();
realm.copyToRealm(friend);
realm.commitTransaction();
```

Como assinalado, é usado o objeto de acesso a dados do framework `Realm` para inserir o objeto `Friend` na base de dados. Isso faz com que não haja dependências entre os resultados dos testes.

Vale reiterar que, devido às informações estarem sendo manipuladas sobre a base de dados pelos métodos de teste da classe `FriendDalTest`, é importante a base de dados ser limpa após a execução de cada método de teste. Isso pode ser feito através de um método com a annotation `@After`, executado após cada método de teste ser executado. Assim, resquícios de informações não serão persistidos entre a execução dos testes.

A execução dos testes de integração é realizada da seguinte forma:

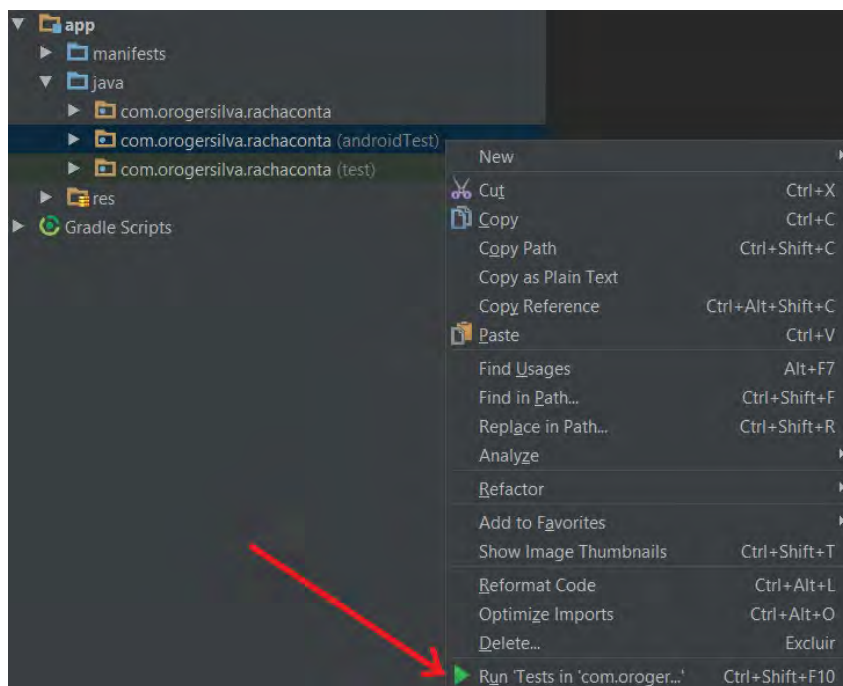


Figura 3.31: Execução de testes de integração em Android

Somente os testes de integração serão executados. E aí vem a pergunta: *é possível executar unit tests e instrumentation tests ao mesmo tempo?* A resposta é: não e sim.

Não é possível executar esses dois tipos de testes através da interface gráfica do Android Studio (até o instante de produção desta publicação). Mas sim, é possível executá-los ao mesmo tempo através do console via linha de comando. Para isso, no console do Android Studio, digite o seguinte comando:

Gradle	Windows	—	gradlew.bat	clean	test
connectedAndroidTest					
Gradle	Linux	—	./gradlew	clean	test
connectedAndroidTest					

Os resultados dos testes não são visíveis pela interface gráfica do Android Studio. Porém, os resultados sobre a execução dos testes são gerados em um arquivo em formato HTML, localizado na seguinte pasta:

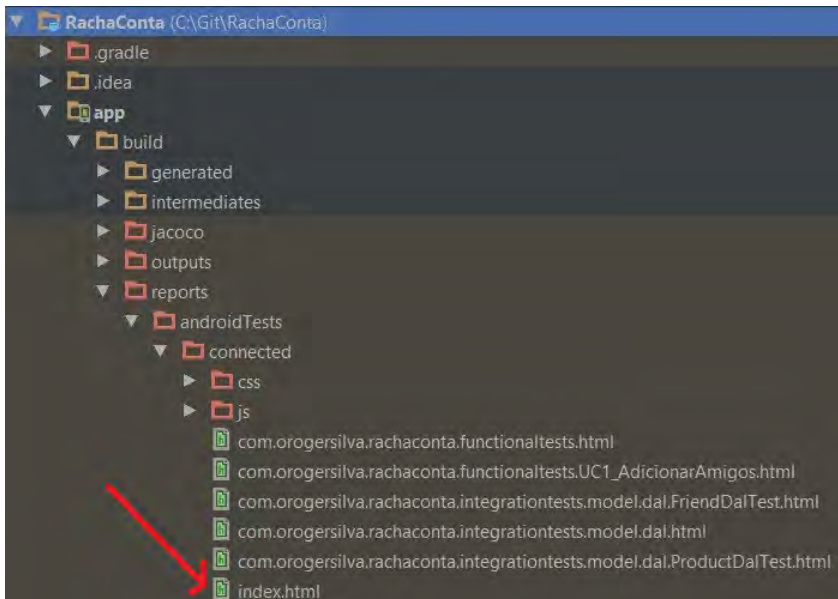


Figura 3.32: Folder com os resultados da execução dos testes de instrumentação

Abrindo esse arquivo em um navegador web, é possível ver os resultados dos testes:

Test Summary



Figura 3.33: Resultado dos testes instrumentados — 1

Packages		Classes			
Package	Tests	Failures	Duration	Success rate	
com.oringersilva.rachaconta.functionaltests	1	0	2.614s	100%	
com.oringersilva.rachaconta.integrationtests.model.dal	7	0	0.878s	100%	

Figura 3.34: Resultado dos testes instrumentados — 2

3.6 TESTES FUNCIONAIS

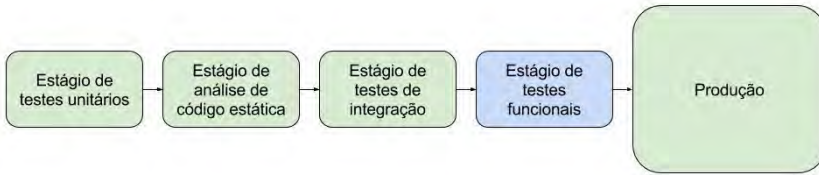


Figura 3.35: Estágio de testes funcionais

Esses testes realizam verificações sobre o software sendo testado, de acordo com a sua especificação. Todos os dados de entrada e saída dos testes baseiam-se exclusivamente na especificação dos casos de uso da aplicação.

A especificação de uma aplicação pode ser expressa por uma abordagem enxuta. Tal abordagem é praticada pela escrita de histórias de usuário, que descrevem necessidades de um usuário sob o ponto de vista dele.

Uma excelente referência que trata sobre como histórias de usuário devem ser escritas é o livro *User Stories Applied: For Agile Software Development*, de Mike Cohn (2004).

Levando-se em conta o aplicativo "Racha Conta", a seguinte história de usuário pode ser escrita para um requisito dessa aplicação:

SENDO um frequentador de *happy hour*

Posso registrar um amigo na lista

Para que possamos rachar a conta ao fim da noite

Toda história de usuário deve ser acompanhada por suas regras de validação. As regras de validação podem ser descritas pela técnica BDD. Através dessa técnica, é possível descrever as validações a serem realizadas sobre uma história de usuário com o uso do pattern **Dado-Quando-Então** (*Given-When-Then*). Uma validação possível para a história anterior pode ser descrita como:

CENÁRIO: Adiciona novo amigo em lista de amigos vazia

Dado que não existem amigos ainda

Quando eu cadastro um novo amigo

Então esse amigo é adicionado na lista

E a lista deixa de ser vazia

A descrição completa de cada validação de cada história de usuário da aplicação será usada para nomear cada método de teste funcional.

Espresso: ferramenta para testes funcionais em Android

É uma ferramenta para testes automatizados de interface de

usuário. Em comparação com ferramentas semelhantes, tem o diferencial de tratar automaticamente da sincronização do fluxo de execução do teste com a UI (*user interface*), sem a necessidade da adição de *sleep's* no código-fonte.

Outro diferencial é a ferramenta ter sido desenvolvida pelo Google. Ou seja, nada melhor do que uma ferramenta desenvolvida para atuar sobre aplicações Android por aqueles que mantêm e conhecem a fundo esse sistema operacional.

A seguir, um método de teste funcional criado para o cenário "*Adiciona novo amigo em lista de amigos vazia*", descrito anteriormente, implementado com recursos do Espresso:

```
@Test
public void dadoQueNaoExistemAmigosAinda_quandoEuCadastroUmNovoA
migo_entaoEsseAmigoEhAdicionadoNaListaEAListaDeixaDeSerVazia() {

    // ARRANGE
    final String FRIEND_NAME = "Roger";

    // ACT
    onView(withId(R.id.deskfloatingactionbutton)).perform(click(
));
    onView(withId(R.id.friend_name_edittext)).perform(typeText(F
RIEND_NAME));
    onView(withId(R.id.add_friend_button)).perform(click());

    // ASSERT
    onView(withId(R.id.friends_recyclerview)).perform(actionOnIt
emAtPosition(0, click()));
}
```

Testes funcionais são tratados como testes de instrumentação (*instrumentation tests*). Assim, é necessário que um emulador Android seja executado antes da execução dos testes.

Espresso é uma ferramenta de testes recomendada para

desenvolvedores, e não testadores. Isso porque ela, por vezes, requer capacidade técnica de customização de algumas classes para viabilizar algumas validações que precisam ser feitas sobre certos componentes de UI mais complexos de serem manipulados, tais como `RecyclerView`s.

No método de teste funcional, para fins de demonstrar o poder de Espresso, vale destacar os seguintes comandos:

```
onView(withId(R.id.friend_name_edittext)).perform(typeText(FRIEND_NAME));  
onView(withId(R.id.add_friend_button)).perform(click());
```

Algumas das melhores características de Espresso são legibilidade e clareza em seus comandos, pois eles requerem pouco esforço para o entendimento de suas intenções. O primeiro comando pode ser interpretado da seguinte forma: *"No componente de campo de texto 'Nome', preencha-o com o nome do amigo"*. Já o comando subsequente relata: *"No componente de botão 'Adicionar', pressione-o"*.

A página do *Android Testing Support Library* contém uma boa introdução sobre componentes do framework Espresso. Veja mais em <https://google.github.io/android-testing-support-library/docs/espresso/index.html>.

Também, uma boa fonte de informações sobre Espresso pode ser encontrado no GitHub da desenvolvedora Chiu-Ki Chan, uma *Android Developer Expert*. Para mais informações, acesse <https://github.com/chiuki>.

Uma frase dita com frequência (e, infelizmente, é com bastante

frequência) é: *"Se a intenção dos testes funcionais é reproduzir exatamente o fluxo de interação de um usuário contra uma aplicação, então eles são completos o suficiente para cobrir todos os casos de uso possíveis e imagináveis. Logo, testes unitários e de integração são desnecessários."* **ERRADO!!!** Muito errado mesmo. Cada categoria de testes tem a sua utilidade. Cada categoria complementa as demais categorias. Alguns pontos que reforçam essa tese podem ser listados.

Em caso de falha, é mais difícil a descoberta do erro

A execução de um método de teste funcional pode cobrir tanto o fluxo de diversas telas da aplicação quanto fazer uso de muitas classes responsáveis por manipular regras de negócio. Digamos que a execução desse método de teste falha. A pergunta que se faz é: *quem garante que a causa do erro foi uma interação incorreta sobre um componente de interface de usuário e não um valor inesperado devolvido por um método da camada de regras de negócio?"* Ninguém pode garantir.

Na verdade, quem poderia garantir, ou fazer da busca pela causa raiz da falha do teste funcional mais simples, seria os testes unitários e de integração. Eles têm como característica testar componentes mais granulares da aplicação, tais como as unidades (classes, métodos etc.) e as interações entre essas unidades, funções estas que testes funcionais não possuem por testarem a aplicação em um contexto mais amplo.

Tempo maior para o conserto de um bug

A suíte de um conjunto de testes funcionais pode levar horas para finalizar sua execução. Muitos times de desenvolvimento executam essa suíte de testes pela madrugada, de modo que, ao amanhecer, todos tenham o resultado da execução. Um

desenvolvedor olha para o resultado e percebe que um método de teste funcional em específico falhou. Logo, um conserto é, então, aplicado.

No caso, o método de teste falhado é executado novamente para que seja visto que o conserto aplicado foi suficiente para a correção da aplicação. Caso esse método de teste leve, em média, dez minutos para ser executado por completo e, seja necessário executá-lo novamente diversas outras vezes para outros defeitos detectados através desse teste, quanto tempo o desenvolvedor pode ficar neste ciclo *executa-conserta-executa*?

Testes unitários são rápidos, por definição. Caso fosse possível cobrir o bug analisado por testes unitários, o tempo do ciclo citado anteriormente seria muito menor, permitindo, assim, que o time de desenvolvimento agregue real valor de negócio à aplicação através da correção aplicada e o usuário usufrua do software corrigido o mais breve possível.

Nesta altura do campeonato, é conhecido como construir cada bloco fundamental que auxilia para a construção de um pipeline de deployment para apps Android. Como dispor cada estágio de testes de forma a viabilizar o processo de automatização que leva as alterações de desenvolvedores sobre projetos de app Android de um repositório de código até a publicação desses apps no Google Play, plataforma de distribuição de apps Android do Google?

No capítulo seguinte, serão abordadas as ferramentas para integração e entrega contínua. Através delas, será possível a automatização de todas as etapas de construção e validação sobre apps.

FERRAMENTAS PARA INTEGRAÇÃO E ENTREGA CONTÍNUA

Ok! Adotaremos a prática de entrega contínua a partir de hoje em nossos projetos. Mas qual ferramenta vamos escolher? É nesse momento que muitas dúvidas surgem. Uma vez que os conceitos sobre integração e entrega contínua estejam dominados, a correta escolha pela ferramenta que fará tudo acontecer é, por muitas vezes, tão complicada quanto implementar um pipeline automatizado do início ao fim.

A ferramenta deve ser simples? Deve ter uma UI agradável? É necessária uma customização sobre como os estágios do pipeline devem estar dispostos? Deve estar na nuvem ou em uma máquina física própria (*self-hosted*)? A configuração do build pode ser compartilhada pelo repositório de código remoto? O uso é gratuito ou pago? Todas essas são questões relevantes de se levar em consideração para a escolha da ferramenta.

Dezenas são os softwares disponíveis para integração/entrega contínua. Muitas novas boas soluções estão ainda sendo lançadas (não que as antigas sejam ruins, muito pelo contrário). A decisão sobre a escolha das ferramentas, nesta publicação, foi embasada em quatro critérios:

- Quão sofisticadas são as ferramentas?
- Quão utilizadas pela comunidade de software são as ferramentas?
- Quão atuais são as ferramentas?
- E, principalmente, quão favoráveis para integrar e distribuir aplicativos Android são as ferramentas?

A sofisticação da ferramenta se faz necessária quando diversas versões de um mesmo app devem ser mantidas e entregues para diversos ambientes, como: um ambiente de teste alpha fechado; um ambiente de teste beta aberto; ou o ambiente de produção, ou seja, o Google Play.

A quantidade de usuários de tais ferramentas é outro ponto importante, pois, que seja dita a verdade: para leigos, implantar integração contínua para um projeto de software é difícil de ser feito. Assim, é fundamental existir uma quantidade considerável de pessoas que debatam soluções para problemas que possam existir durante o processo de implantação. Também é importante a ferramenta não ser ultrapassada, pois qual é a garantia do suporte que será dado a ela nos próximos anos (ou até meses, sob uma ótica pessimista)?

E, finalmente, o que realmente importa para nós: deve ser, no mínimo, viável montar o pipeline de entrega para projetos de software Android. A ferramenta deve permitir invocar tasks do Gradle, iniciar a execução do emulador para testes funcionais, e permitir a publicação do aplicativo de forma automatizada no Google Play.

A seguir, veja as ferramentas para integração contínua a serem tratadas por este livro. A apresentação delas se dá por ordem de sofisticação: da mais simples até a mais sofisticada.

4.1 TRAVIS CI

O Travis CI pode ser acessado em <https://travis-ci.org/>.

Trata-se de um serviço de integração contínua hospedado e distribuído na nuvem usado para realizar builds, testes e deploys sobre projetos hospedados no GitHub. Projetos presentes no GitHub podem ser processados pelo Travis CI gratuitamente, desde que a execução seja limitada a um *job* por vez.

Um *job* é uma entidade executável controlada pela ferramenta de integração contínua, usada para construir e testar software.

A tela principal do Travis CI, para um determinado projeto, é exibida da seguinte forma:



Figura 4.1: Tela principal do Travis CI

Como demarcadas na figura, as seções da tela anterior podem

ser descritas da seguinte forma (conforme os índices sobre cada seção):

1. Nome do repositório compartilhado no GitHub da aplicação a ser construída e validada.
2. Status do build realizado sobre a aplicação contendo descrição da mensagem de commit, rótulo do build, tempo gasto para execução do build, autor do commit e branch sobre a qual o commit buildado pertence.
3. Status resumido sobre os últimos builds realizados sobre projetos do usuário da conta.
4. Log de execução do build.

Construindo o pipeline

Para que uma aplicação seja construída pelo Travis CI, deve ser adicionado no diretório raiz do projeto da aplicação um script de build nomeado como `.travis.yml`.

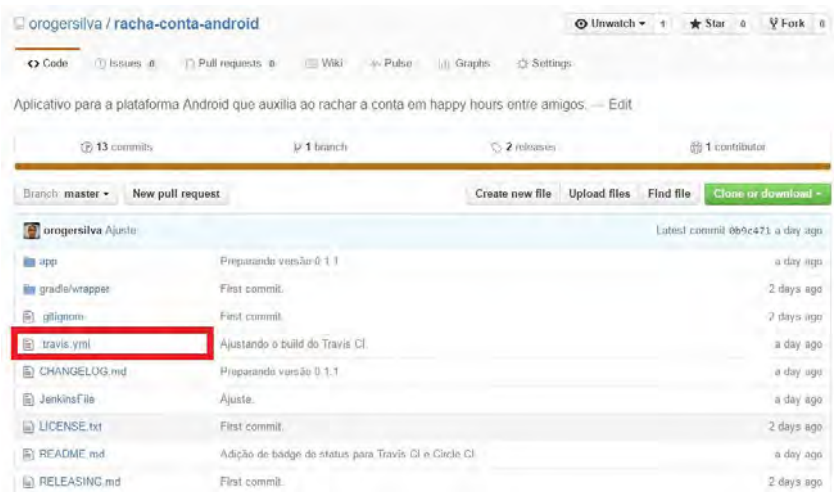


Figura 4.2: Script de build no diretório raiz do projeto

O `.travis.yml` trata-se de um arquivo do tipo YAML (YAML

Ain't Markup Language). Esse arquivo pode ser definido como:

```
language: android
android:

components:
  # Descomente as linhas abaixo se você quiser
  # usar a última revisão de Android SDK Tools
  - platform-tools
  - tools

  # A versão do BuildTools usada pelo seu projeto
  - build-tools-23.0.2

  # A versão do SDK Android usada para compilar seu projeto
  - android-23

  # Componentes adicionais
  - extra-google-google_play_services
  - extra-google-m2repository
  - extra-android-m2repository
  - addon-google_apis-google-19

  # Especifique, no mínimo, uma imagem de sistema
  # se você precisar executar o emulador durante seus testes
  - sys-img-armeabi-v7a-android-19
  - sys-img-x86-android-17

before_install:
  - chmod +x gradlew

# Emulador Management: Create, Start and Wait
before_script:
  # - echo no | android create avd --force -n test -t android-19 --abi armeabi-v7a
  # - emulator -avd test -no-skin -no-audio -no-window &
  # - android-wait-for-emulator
  # - adb shell input keyevent 82 &

script:
  ./gradlew clean testRelease lintRelease connectedReleaseAndroidTest assembleRelease publishApkRelease
```

Figura 4.3: Script de build para o Travis CI

Alguns trechos desse arquivo merecem destaque:

```
language: android
```

Esse trecho permite ao Travis CI saber em qual tipo de ambiente deve ser construída a aplicação. No caso desse exemplo, deve ser instanciado um ambiente Android.

Uma vez instanciado um ambiente adequado para o build do projeto Android, componentes do *SDK tools* e *platform-tools* devem estar disponíveis. Essa disponibilidade será viabilizada

automaticamente pelo script de build, como segue:

```
components:
  - platform-tools
  - tools
```

Para a execução do build da aplicação, são necessários os *build-tools* (ferramentas de build):

```
- build-tools-23.0.2
```

Mas qual é a versão do SDK que o script de build deve obter? Isso também deve estar explícito no script. Preferencialmente, é recomendado que o script obtenha a última versão estável do SDK:

```
- android-23
```

Tendo em vista que serão executados testes de instrumentação, faz-se necessário que um emulador seja disparado. Logo, o script de build também deve conter o setup para essa ação:

```
- echo no | android create avd --force -n test -t android-19 --abi armeabi-v7a
- emulator -avd test -no-skin -no-audio -no-window &
- android-wait-for-emulator
- adb shell input keyevent 82 &
```

Por fim, as tasks que refletem a execução dos estágios do pipeline de deployment:

```
./gradlew clean testRelease lintRelease connectedReleaseAndroidTest assembleRelease publishApkRelease
```

Cada task pertencente ao script cobre um ou mais estágios do pipeline.

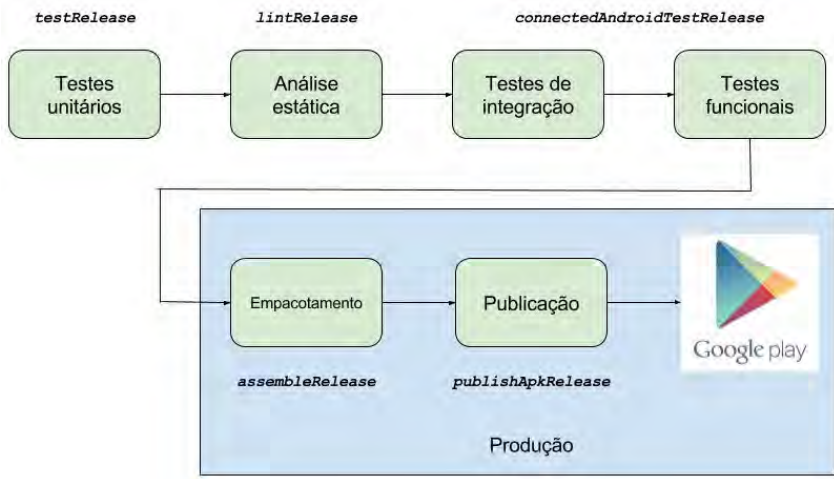


Figura 4.4: Pipeline de deployment com tasks

Com a intenção de avançar sobre o entendimento de separação de responsabilidades sobre um pipeline de deployment, o estágio *Produção* foi separado em outros dois estágios: *Empacotamento* e *Publicação*. O primeiro é o responsável por gerar o apk assinado do aplicativo e disponibilizá-lo para o segundo, que será o responsável por publicá-lo no Google Play.

Uma vez que as configurações para o build estejam descritas no arquivo `.travis.yml` e elas estejam na raiz do projeto da aplicação, o Travis CI pode construir, validar e publicar o aplicativo. Da máquina do desenvolvedor ao Google Play, o seguinte percurso é realizado:

1. O desenvolvedor executa por meio da ferramenta de controle de versão o envio de todas as modificações sobre arquivos do projeto da aplicação para o GitHub (`git push`).
2. O GitHub notifica o Travis CI que alterações sobre o repositório do projeto do aplicativo Android foram realizadas.
3. É executado o script contido no arquivo `.travis.yml` sobre o projeto da aplicação.

4. É disponibilizado um status com o resultado do processamento do script.
5. (Opcional) É enviado um e-mail para *stakeholders* caso o build tenha sido quebrado ou consertado.

```
BUILD SUCCESSFUL

Total time: 1 mins 50.3 secs

The command "./gradlew clean testRelease assembleRelease" exited with 0.

Done. Your build exited with 0.
```

Figura 4.5: Build realizado com sucesso no Travis CI

```
BUILD FAILED

Total time: 42.91 secs

The command "./gradlew clean testReleaseUnitTestCoverage assembleRelease" exited with 1.

Done. Your build exited with 1.
```

Figura 4.6: Falha no build do Travis CI



orogersilva / racha-conta-android (master)



Build #1 failed.



2 minutes and 29 seconds



Roger Silva

255c3dd Changeset —

Configurando deployment continuo.

Want to know about upcoming build environment updates?

Would you like to stay up-to-date with the upcoming [Travis CI](#) build environment updates? We set up a mailing list for you! Sign up [here](#).



[Documentation](#) about Travis CI

For help please join our IRC channel [irc.freenode.net#travis](#).

Choose who receives these build notification emails in your [configuration file](#).

Would you like to test your private code?

[Travis CI for Private Projects](#) could be your new best friend!



Figura 4.7: Notificação de falha no build do Travis CI

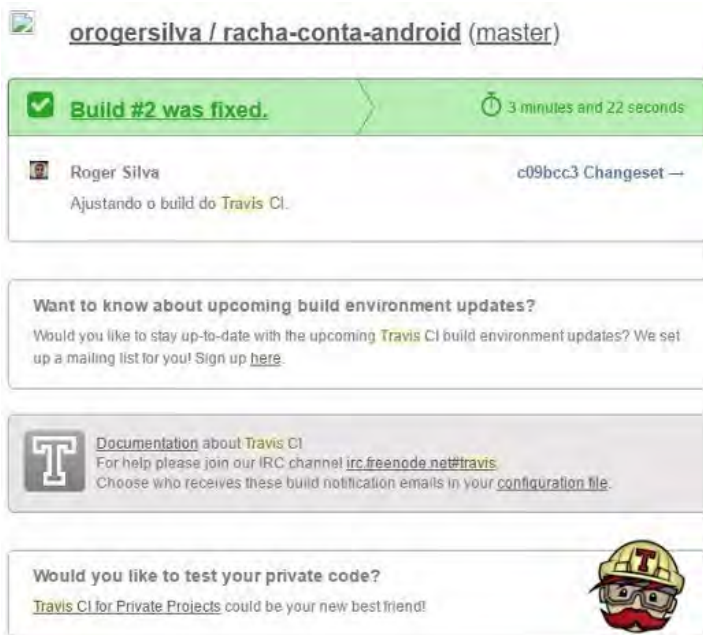


Figura 4.8: Notificação de conserto no build do Travis CI

É de fundamental importância o envio de algum tipo de notificação a partes interessadas em caso de quebra e conserto de builds em servidores de integração. Isso permite que qualquer comportamento fora do comum durante a integração da aplicação (em caso de quebra no build) seja imediatamente sinalizado e, da mesma forma, que seja notificado quando o build é estabilizado (em caso de conserto do build).

4.2 GOCD

É uma ferramenta open source usada para entrega contínua de software. O uso na implantação de projetos com a prática de entrega contínua é gratuito. Porém, caso seja necessário suporte técnico especializado, esse serviço será cobrado.

O GoCD pode ser obtido em <https://www.go.cd/>.

Para o uso da solução GoCD, é necessário realizar o download de dois instaladores: o `agent` e o `server`. O `agent` GoCD é responsável por processar todas as operações relacionadas ao correto funcionamento dos pipelines. Já o `server` GoCD é quem delega todas as operações para que o `agent` GoCD realmente as execute.

Um `server` pode delegar que um processamento seja realizado por mais de um `agent` ao mesmo tempo, caso esse seja um processamento possível de ser realizado concorrentemente.

Uma vez que o `server` e `agent` sejam instalados, o GoCD pode ser acessado através da URL: <http://localhost:8153>. Ao acessá-lo, o usuário da ferramenta se depara com a seguinte tela:

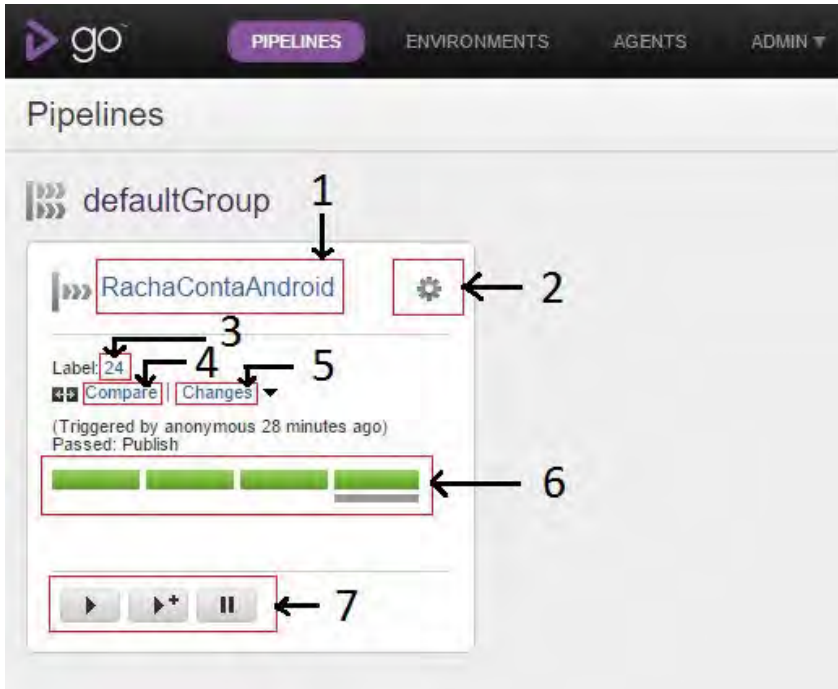


Figura 4.9: Tela principal do GoCD

A figura anterior exibe informações sobre o build realizado em um pipeline.

1. Nome do pipeline.
2. Acesso ao painel de configurações do pipeline.
3. Rótulo do último build realizado sobre o pipeline.
4. Acesso a um painel de comparação sobre builds — especificamente, o penúltimo e último builds executados.
5. Últimas alterações realizadas sobre o projeto da aplicação.
6. Estágios do pipeline com status de sucesso ou falha.
7. Botões para iniciar um build padrão, build customizado e pausar o build corrente, respectivamente.

Na figura apresentada, já existe um pipeline criado com nome de `RachaContaAndroid`. Antes de apresentar o passo a passo para

a criação desse pipeline, deve ser salientado que GoCD é uma ferramenta de gerenciamento de workflows muito sofisticada. Em consequência disso, os seguintes conceitos pertencentes ao domínio da ferramenta devem ser compreendidos.

- **Task:** é um simples comando. Preferencialmente, esse comando deve ter somente uma responsabilidade. Isso porque, em caso de falha, será simples de detectar sua causa em um contexto mais amplo. No caso do GoCD, seria durante a execução do processamento de todos os estágios do pipeline.
- **Job:** é um conjunto de *tasks* que são executadas sequencialmente. *Tasks* pertencentes ao mesmo *job* serão executadas pelo mesmo *agent*.
- **Stage:** é um conjunto de *jobs* que podem ser executados concorrentemente. Ou seja, podem ser executados por vários *agents* ao mesmo tempo.
- **Pipeline:** é um conjunto de *stages* que são executados sequencialmente.
- **Material:** ele inicia a execução do *pipeline*. Normalmente, costuma ser um repositório Git quem faz o papel de *material*, mas poderia ser a disponibilização de um artefato gerado, ou mesmo outro *pipeline*.

Com base nos conceitos apresentados referentes ao GoCD, segue o projeto do *pipeline* `RachaContaAndroid` :

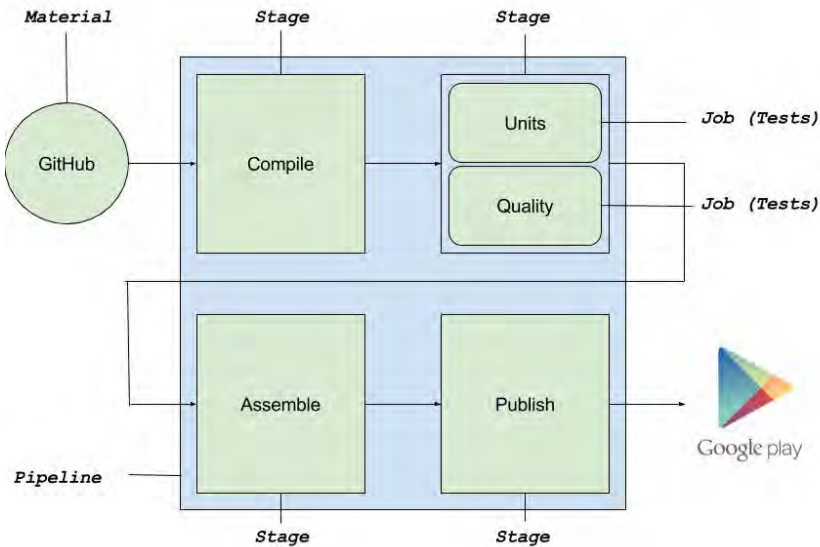


Figura 4.10: Projeto de pipeline de deployment no GoCD

Construindo o pipeline

Para a criação desse pipeline, deve ser acessada, via navegador, a URL <http://localhost:8153>. Após, selecionar no menu do GoCD a opção Admin > Pipelines . Por fim, criar o *pipeline*, como mostrado a seguir:

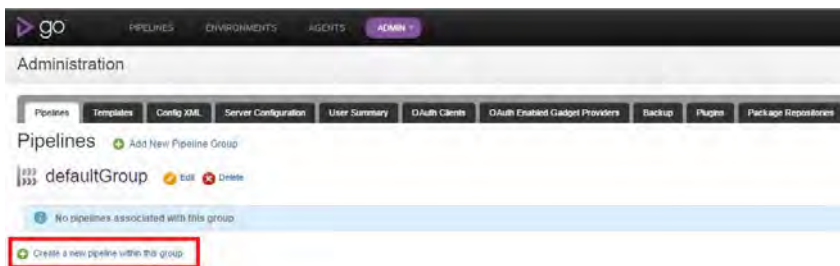


Figura 4.11: Criação do pipeline no GoCD

Então, o *pipeline* deve ser nomeado:

The screenshot shows the 'Add Pipeline' interface. At the top, there's a navigation bar with 'go' logo and links for 'PIPELINES', 'ENVIRONMENTS', 'AGENTS', and 'ADMIN'. Below this is a header 'Add Pipeline'. The main content area has three tabs: 'Step 1: Basic Settings', 'Step 2: Materials', and 'Step 3: Stage/Job'. The 'Basic Settings' tab is active. It contains two text input fields: 'Pipeline Name*' with the value 'RachaContaAndroid' and 'Pipeline Group Name' with the value 'defaultGroup'. A note below the fields states '* indicates a required field'.

Figura 4.12: Nomeação do pipeline

A seguir, detalhes sobre o *material* devem ser preenchidos. O tipo de *material* usado pelo *pipeline* é um repositório Git, o qual acionará a execução do *pipeline* quando um novo push for realizado sobre o repositório. Também deve ser informada a URL do repositório Git e a branch sobre a qual devem ser enviadas alterações no projeto do aplicativo, para que o *pipeline* seja ativado.

The screenshot shows the 'Add Pipeline' interface in the 'go' tool. The top navigation bar includes 'PIPELINES', 'ENVIRONMENTS', 'AGENTS', and 'ADMIN'. The main heading is 'Add Pipeline'. Below it, there are three steps: 'Step 1: Basic Settings', 'Step 2: Materials' (which is the active step), and 'Step 3: Stage/Job'. The 'Materials' section contains the following fields and options:

- Material Type***: A dropdown menu with 'Git' selected.
- URL***: A text input field containing 'https://github.com/orogersilva/racha-conta-ai'.
- Branch**: A text input field containing 'master'.
- ☒ Poll for new changes
- ☐ Shallow clone (recommended for large repositories)
- CHECK CONNECTION**: A button.
- Connection OK.

Figura 4.13: Definição do material

A próxima etapa é iniciar a configuração do primeiro *stage* e *job*. O *stage* *Compile* contém jobs responsáveis por compilar o projeto da aplicação. No caso, esse responsável será o *job* *Compile*.

go

PIPELINES

ENVIRONMENTS

AGENTS

ADMIN

Add Pipeline

Step 1:
Basic Settings

Step 2:
Materials

Step 3:
Stage/Job

Stage/Job

Configuration Type ☒ Define Stages ☐ Use Template

Stage Name*

Compile

Trigger Type: ☒ On Success ☐ Manual ?

* indicates a required field

Initial Job and Task

You can add more jobs and tasks to this stage once the stage has been created.

Job Name*

Compile

Task Type*

Gradle

Tasks:*

clean compileReleaseSources

Figura 4.14: Definição de stage e job

O GoCD suporta a execução de tasks Gradle através da instalação de um plugin. O arquivo .jar referente ao plugin pode ser acessado em <https://github.com/jmnarloch/gocd-gradle-plugin/releases>. Em seguida, o plugin deve ser armazenado no folder \$GO_SERVER_HOME/plugins/external (\$GO_SERVER_HOME é o caminho no sistema de arquivos onde está instalado o server do GoCD).

Neste ponto, já é possível executar o *pipeline* RachaContaAndroid . Porém, ele somente compilará o projeto do aplicativo. Assim, continuando a construção do *pipeline*, será criado o *stage* Tests pelo painel de configurações do *pipeline*.

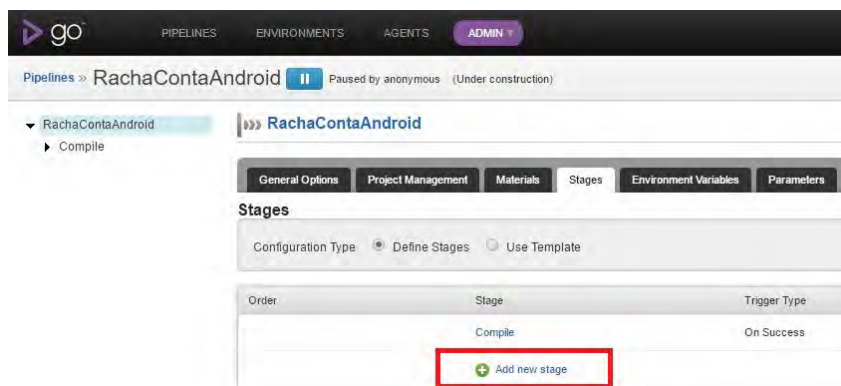


Figura 4.15: Criação de novo stage

No *stage* Tests , será definido o tipo de trigger (On Success ou Manual) que indica se o próximo *stage* será iniciado automaticamente, ou se deverá ser iniciado de forma manual pelo operador do GoCD. No mesmo diálogo, deve ser definido o nome do primeiro *job* desse *stage*, no caso, o *job* Units (este responsável pela execução de testes unitários).

The screenshot shows the 'Add Stage' dialog box with the 'Stage Information' tab selected. The 'Stage Name*' field contains the text 'Tests'. The 'Trigger Type:' section has two radio buttons: 'On Success' (which is selected) and 'Manual'. A small question mark icon is next to the 'Trigger Type:' label. At the bottom, a note states '* indicates a required field'.

Figura 4.16: Definição do stage Tests — 1

The screenshot shows the 'Add Stage' dialog box with the 'Initial Job and Task' tab selected. A message at the top reads: 'You can add more jobs and tasks to this stage once the stage has been created.' The 'Job Name*' field contains 'Units'. The 'Task Type*' dropdown menu is set to 'Gradle'. The 'Tasks:*' field contains the text 'testReleaseUnitTest'.

Figura 4.17: Definição do stage Tests — 2

O *stage* `Tests`, porém, deve conter o *job* responsável por realizar a análise estática de código sobre o projeto da aplicação. Esse será o *job* `Quality`. A criação do novo *job* para o *stage* `Tests`, já criado, deve ser realizada da seguinte forma:

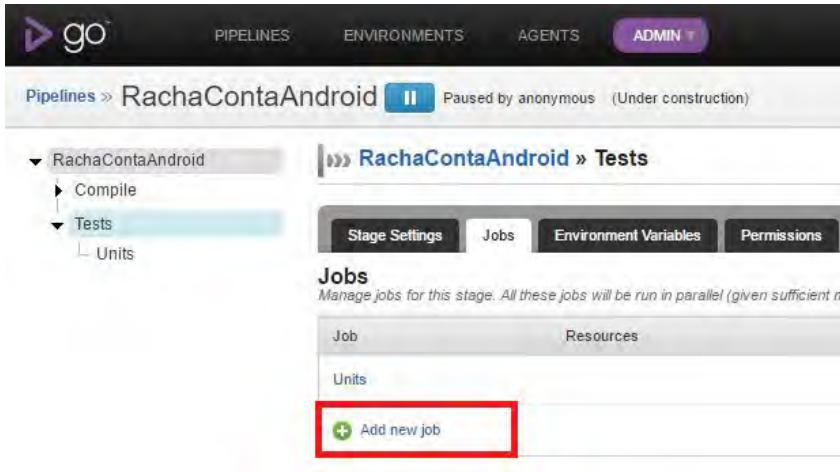


Figura 4.18: Criação de novo job

A criação de cada *stage* e *job* restantes se dá de forma similar aos já criados. As *tasks* executadas pelo *pipeline* são exatamente as mesmas usadas pelo Travis CI.

Uma vez que todos os componentes do *pipeline* estejam definidos, ele deverá assumir a seguinte estrutura:

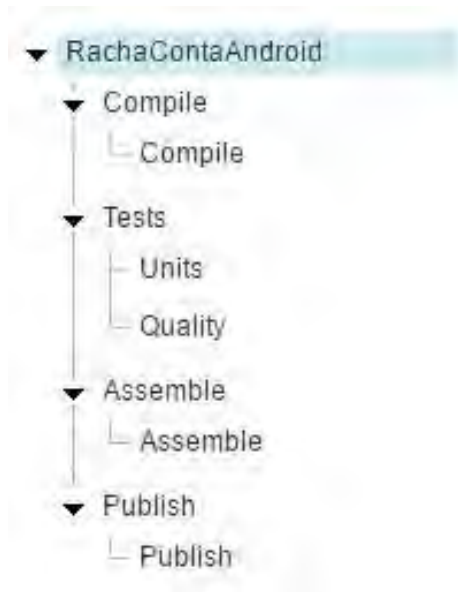


Figura 4.19: Esqueleto do pipeline RachaContaAndroid

Com o *pipeline* pronto, a cada vez que forem comitadas alterações no repositório no GitHub (no contexto do GoCD, o *material*), o *pipeline* será executado automaticamente para construir, validar e publicar o app Android para o Google Play.

No arquivo `build.gradle`, em nível de módulo, estão descritas instruções como estas:

```
storePassword System.getenv("KEYSTORE_PASSWORD")
keyAlias System.getenv("ALIAS_NAME")
keyPassword System.getenv("ALIAS_PASSWORD")
```

Essas instruções acessam variáveis de ambiente. Informações sigilosas, como por exemplo, senha de chave criptografada usada para assinar o `apk` do aplicativo, não devem estar explícitas no `build.gradle`. Dados como esses devem ser setados nas variáveis de ambiente do software de integração

contínua, como o Travis CI e GoCD. Dessa forma, uma vez que o script de build seja processado por servidores de integração, o dado sigiloso será obtido pelo script diretamente do servidor de integração, onde estará protegido, tal como mostrado a seguir na configuração do *pipeline* do GoCD:

The screenshot displays the GoCD configuration interface with two tabs: "Environment Variables" and "Secure Variables". The "Environment Variables" tab is active, showing a table with columns "Name" and "Value". It contains three entries: "ANDROID_HOME" with value "C:\Android\SDK", "JAVA_HOME" with value "C:\Program Files\Java\jdk1.8.0_72", and an empty row. Below the table is an "Add" button. The "Secure Variables" tab is also visible, showing a similar table with four entries: "ALIAS_NAME", "ALIAS_PASSWORD", "KEYSTORE_PASSWORD", and "SERVICE_ACCOUNT_EMAIL", all with masked values (dots). Each entry has an "Edit" button and a delete icon. The "Secure Variables" section is highlighted with a red border.

Name	Value
ANDROID_HOME	C:\Android\SDK
JAVA_HOME	C:\Program Files\Java\jdk1.8.0_72

+ Add

Name	Value	Edit
ALIAS_NAME	*****	Edit
ALIAS_PASSWORD	*****	Edit
KEYSTORE_PASSWORD	*****	Edit
SERVICE_ACCOUNT_EMAIL	*****	Edit

+ Add

Figura 4.20: Definição de variáveis de ambiente

O exemplo da imagem anterior, vale também para qualquer servidor de integração contínua.

4.3 JENKINS

Um dos softwares de integração contínua mais conhecidos pela comunidade de desenvolvimento de software é o Jenkins. Uma de suas principais características é sua extensibilidade por meio do uso de plugins.

O Jenkins pode ser obtido em <https://jenkins.io/>. É fortemente recomendável o uso de uma versão estável. Também está disponível para download versões não estáveis. Porém, nesse último caso, muitas funcionalidades podem apresentar comportamento inesperado.

Ao finalizar a instalação do Jenkins, ele será iniciado automaticamente no navegador padrão. Caso contrário, para acessar seu painel principal, deve ser digitada a seguinte URL no navegador: <http://localhost:8080>. O navegador deverá mostrar a seguinte tela:



Figura 4.21: Painel principal do Jenkins

Plugins do Jenkins

Como mencionado anteriormente, Jenkins suporta uma infinidade de plugins que permitem sua extensibilidade. Logo, para a construção de um pipeline de deployment para o aplicativo *Racha Conta*, alguns plugins deverão ser instalados. Dentre eles, alguns

voltados à plataforma Android.

Para a instalação dos plugins, deve ser acessada, a partir do painel principal do Jenkins, a opção de menu Gerenciar Jenkins > Gerenciar plugins. A seguir, a aba Disponíveis deve ser selecionada para que, então, os plugins possam ser filtrados através do campo de busca. Dentre os plugins a serem instalados, estão:

- **Android Emulator Plugin:** inicialização e execução de um aplicativo em um emulador Android (AVD — *Android Virtual Device*).
- **Android Lint Plugin:** realização de análise estática de código sobre o projeto da aplicação Android.
- **Build Pipeline Plugin:** exibição visual de um pipeline e o disparo manual de cada estágio desse pipeline, caso necessário.
- **Checkstyle Plugin:** realização de análise estática de código sobre o projeto da aplicação Android.
- **FindBugs Plugin:** realização de análise estática de código sobre o projeto da aplicação Android.
- **GitHub Plugin:** permite a integração do GitHub ao Jenkins.
- **Gradle Plugin:** permite a execução de scripts Gradle.
- **Jacoco Plugin:** geração de report de análise de cobertura de código e amostragem visual dos resultados coletados.
- **PMD Plugin:** realização de análise estática de código sobre o projeto da aplicação Android.
- **xUnit Plugin:** geração de report de execução de testes unitários e exibição visual de seus resultados.

Duas observações merecem ser feitas sobre os plugins. Sobre os plugins relativos à análise estática de código, vale relembrar que eles têm funções complementares. Ou seja, cada mecanismo realiza um tipo de análise sobre o código, de forma que a adoção desses plugins em conjunto faz-se plausível.

Por fim, sobre o plugin para emular um dispositivo Android, uma coisa precisa ser dita: ele é problemático. Apesar de ser o plugin mais recomendável para emulação de dispositivos Android, conforme novas versões são lançadas, algumas características do plugin que, em versões anteriores funcionavam normalmente, podem passar a deixar de funcionar.

Dependências de sistema operacional (Linux, Windows etc.) e sobre a máquina ser 32 ou 64 bits também são elementos difíceis de serem tratados adequadamente pelo plugin. Logo, a execução de testes funcionais sobre apps Android pode ser dificultada no Jenkins. A sua viabilidade dependerá muito de experimentos durante a configuração desse plugin.

Todavia, para fins de demonstrar a configuração do estágio de testes funcionais no Jenkins, será usada uma versão estável desse plugin, mesmo não sendo a versão mais atual. E uma boa fonte de informações sobre esse plugin é seu GitHub (<https://github.com/jenkinsci/android-emulator-plugin>).

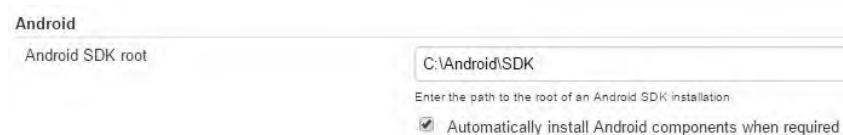
A definição da variável de ambiente `ANDROID_HOME` também é necessária no Jenkins, para que ele conheça onde está localizado o SDK Android. Para isso, a partir do seu painel principal, navegue através de `Gerenciar Jenkins > Configurar o sistema`. Então, defina a variável de ambiente `ANDROID_HOME` e também o

valor para o campo *Android SDK root*.



Variáveis de ambiente	
Lista de pares de chave-valor	
nome	ANDROID_HOME
valor	C:\Android\SDK

Figura 4.22: Definição da variável de ambiente ANDROID_HOME



Android

Android SDK root

C:\Android\SDK

Enter the path to the root of an Android SDK installation

☒ Automatically install Android components when required

Figura 4.23: Definição do caminho para a pasta do SDK Android

Construindo o pipeline

O pipeline construído no Jenkins conterá estágios com as mesmas responsabilidades que os estágios usados para a construção do pipeline no GoCD.

Os estágios do pipeline no Jenkins são construídos através de jobs. Assim, na tela principal do Jenkins, deve ser selecionado **Novo job** e, então, o job nomeado como `RachaContaAndroidCompile`.

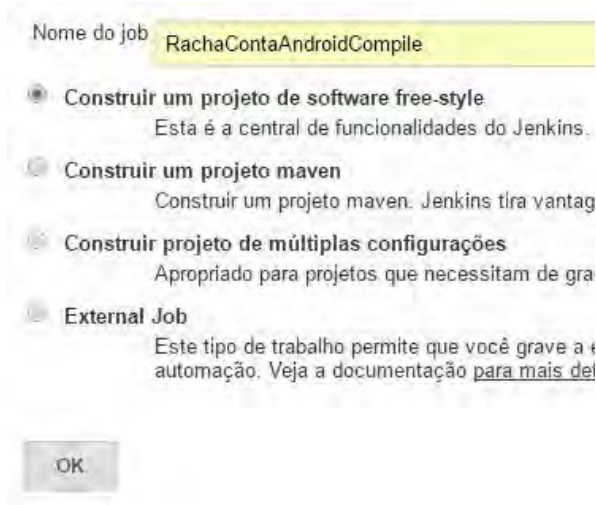


Figura 4.24: Definindo o job Compile

Algumas configurações do job são triviais, tais como nome do job e descrição. Outras, porém, merecem destaque.

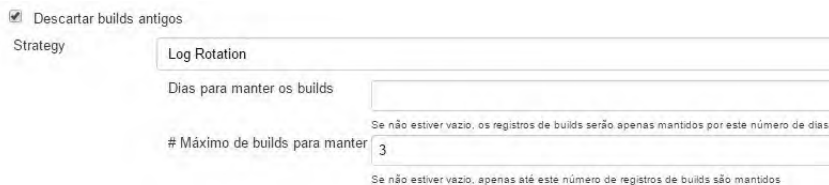


Figura 4.25: Descartando builds antigos

Essa configuração determina que somente serão persistidos logs, relatórios e artefatos resultantes de builds durante um determinado período de tempo ou somente de um certo número de builds. Como mostrado na figura, somente os três últimos builds do job terão suas informações persistidas, o que auxilia no gerenciamento de espaço em disco da máquina que hospeda a ferramenta.

A seguir, deve ser definido o endereço do repositório de gerenciamento de código-fonte, o qual o Jenkins obterá o projeto do

aplicativo a ser construído.



Gerenciamento de código fonte

☐ Nenhum

☐ CVS

☐ CVS Projectset

☒ Git

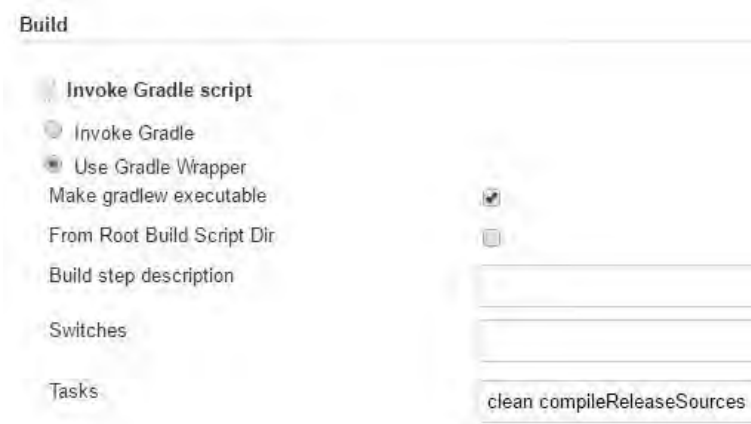
Repositories

Repository URL

Credentials

Figura 4.26: Configurando URL do repositório remoto

Uma vez que o projeto seja adquirido do sistema de controle de versão para a máquina na qual o Jenkins está hospedado, o processo de compilação poderá ser executado. Para defini-lo, selecione Adicionar passo de build e, então, a opção para invocar o script Gradle. Uma nova subseção de formulário estará disponível. Nela, devem ser descritas quais tasks do Gradle serão executadas por este estágio.



Build

☒ Invoke Gradle script

☐ Invoke Gradle

☒ Use Gradle Wrapper

Make gradlew executable ☒

From Root Build Script Dir ☐

Build step description

Switches

Tasks

Figura 4.27: Definindo tasks Gradle para estágio Compile

A importância de um estágio de compilação em um pipeline de integração contínua, além de compilar a aplicação, é fornecer rápido feedback sobre falhas triviais.

Por fim, basta salvar as configurações do job. Com as configurações salvas, o projeto *Racha Conta* pode ser construído.

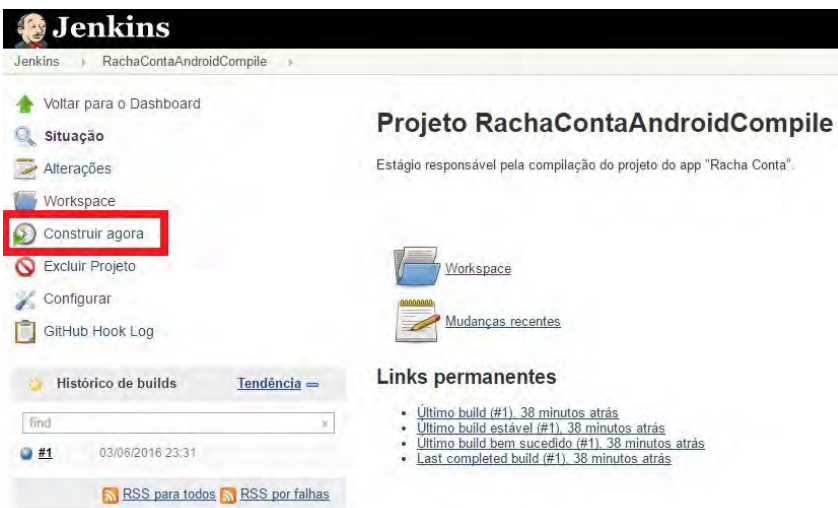


Figura 4.28: Compilando projeto Racha Conta

Caso o job tenha sido executado com sucesso, seu status atual será exibido na tela principal do Jenkins com o ícone de um sol.

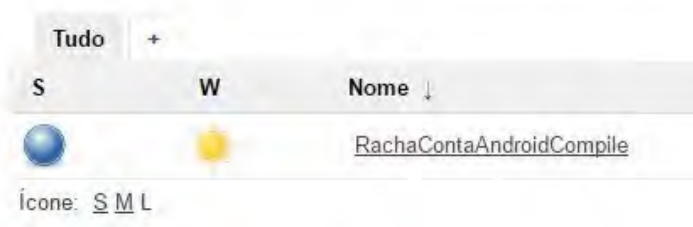


Figura 4.29: Status do job Compile

Caso contrário (com uma falha na execução do build), um ícone em formato de chuva com trovoadas será mostrado.

Seguindo com o próximo job a ser configurado, que trata da execução de testes unitários, este será nomeado como `RachaContaAndroidUnits`. Sua configuração em muito se assemelha ao job anterior. Logo, serão mencionados somente os pontos relevantes para essa configuração.

Na seção `Opções avançadas do projeto`, deve ser selecionada a opção `Usar workspace customizado`. Uma vez que o projeto da aplicação é obtido do repositório de código remoto pelo job `RachaContaAndroidCompile`, ele é armazenado em um diretório na máquina local denominado de *workspace*. Para que não seja necessário repetir esse procedimento, na configuração do job `RachaContaAndroidUnits` (e nas configurações dos próximos jobs), o projeto do aplicativo *Racha Conta* será referenciado para esse *workspace* já definido.

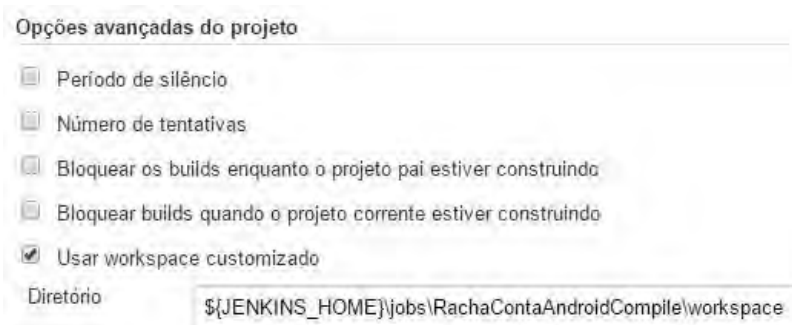


Figura 4.30: Referenciando workspace já existente

Em seguida, deve ser adicionado um passo de build com a invocação de um script Gradle. Nele, deve ser descrita a task responsável pela execução de testes unitários.



Figura 4.31: Descrevendo task de execução de testes unitários

Adicionalmente, deve ser configurada uma ação pós-build. Essa ação é executada uma vez que o passo de build tenha sido finalizado, ou seja, a task `testReleaseUnitTest`. A ação a ser selecionada é `Publicar relatório de testes do JUnit`. Após a execução dos testes unitários sobre o aplicativo, é gerado um relatório contendo detalhes dos resultados alcançados com a execução dos testes unitários.

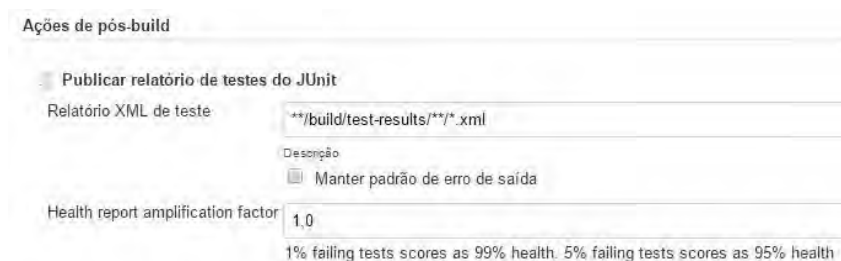


Figura 4.32: Configurando ação de pós-build

Com o relatório disponível, o Jenkins vai analisá-lo e, então, exibirá informações sobre os resultados na forma de um gráfico.

Dentre as informações, estão o número de testes unitários executados e, dentre eles, quantos passaram e falharam.

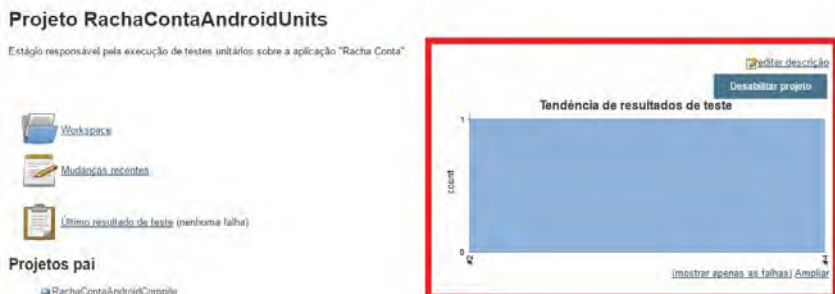


Figura 4.33: Gráfico de resultados de testes unitários

Com dois jobs configurados (`RachaContaAndroidCompile` e `RachaContaAndroidUnits`), já é possível encadear a execução dos estágios e visualizar esse encadeamento na forma de um pipeline de deployment. Assim, voltando à configuração do job `RachaContaAndroidCompile`, deve ser definida uma ação pós-build que permita o trigger de build parametrizados. No caso, o trigger não será parametrizado, e sim, será uma simples inicialização da execução do estágio de testes unitários quando o estágio de compilação tiver sido finalizado com sucesso. Essa ação pós-build deve ser definida como a seguir:

Ações de pós-build

Trigger parameterized build on other projects

Build Triggers

Projects to build:

Trigger when build is:

Trigger build without parameters ☒

Figura 4.34: Encadeando execução de estágios em um pipeline

Tendo sido disparada a execução do estágio `RachaContaAndroidCompile` , em caso de execução com sucesso, automaticamente o estágio `RachaContaAndroidUnits` será executado. Isso de forma que a aplicação não somente seja verificada quanto à sua correta compilação, como também quanto à validação dos seus módulos através de testes unitários.

Essa configuração de encadeamento não somente é válida para os estágios `RachaContaAndroidCompile` e `RachaContaAndroidUnits` , como também entre os estágios:

`RachaContaAndroidUnits` e `RachaContaAndroidQuality` ,

`RachaContaAndroidQuality` e `RachaContaAndroidInstrumented` ,

`RachaContaAndroidInstrumented` e `RachaContaAndroidAssemble` ,

`RachaContaAndroidAssemble` e `RachaContaAndroidPublish` .

O próximo estágio a ser configurado é aquele responsável pela análise estática de código, através do job `RachaContaAndroidQuality` . Em relação às suas configurações, ele se assemelha ao job configurado anteriormente. O que distingue um do outro são as tasks responsáveis pela análise estática e a configuração de publicação dos relatórios com os resultados dessas análises. Elas se dão da seguinte forma:

Build

☒ **Invoke Gradle script**

☐ Invoke Gradle

☒ Use Gradle Wrapper

Make gradlew executable ☒

From Root Build Script Dir ☒

Build step description

Switches

Tasks

Figura 4.35: Descrevendo tasks de análise estática de código

Ações de pós-build

☒ Publish Android Lint results

Lint files

[Fileset includes](#) setting that specifies the generated Lint XML

☒ Publish Checkstyle analysis results

Checkstyle results

[Fileset includes](#) setting that specifies the generated

☒ Publish FindBugs analysis results

FindBugs results

[Fileset includes](#) setting that specifies the generated reports, respectively. Be sure not to include any non-report

Use rank as priority ☐

Uses the bug rank when evaluating the priority

☒ Publish PMD analysis results

PMD results

[Fileset includes](#) setting that specifies the generated reports

Figura 4.36: Configuração da publicação dos resultados das análises de código

Da mesma forma que na publicação de resultados do relatório de testes unitários no estágio `RachaContaAndroidUnits`, é gerada uma publicação semelhante após a execução do estágio

RachaContaAndroidQuality .

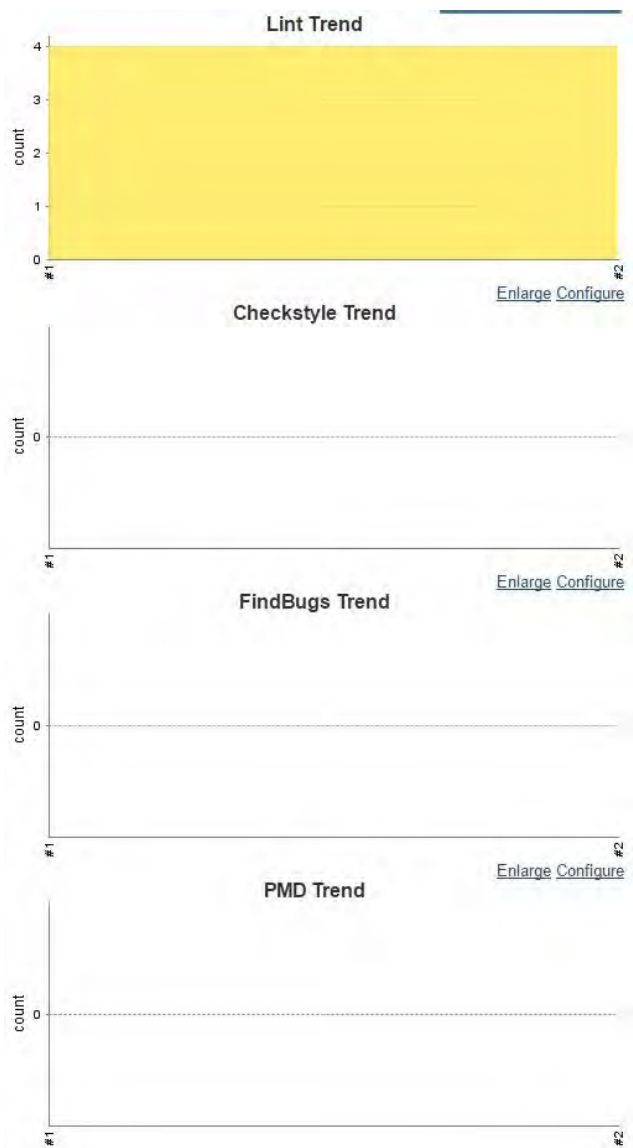


Figura 4.37: Publicação dos resultados das análises de código

São gerados gráficos para os resultados da análise estática de código de cada ferramenta de análise. Na figura mostrada, é possível

ver que a ferramenta Lint detectou quatro issues. Já as outras ferramentas não detectaram issues nas análises realizadas.

O estágio seguinte a ser preparado é encarregado pela validação das funcionalidades disponibilizadas pelo app. Esse estágio, que executará uma bateria de testes funcionais sobre a aplicação, se chamará `RachaContaAndroidInstrumented` (o nome origina-se da expressão "testes de instrumentação").

A configuração do job referente a esse estágio tem uma particularidade: seu build é parametrizado. Ou seja, a execução do job necessita de informações previamente determinadas na configuração do job. Os valores fornecidos devem ser preenchidos pelo responsável pela configuração do job, pois, para a geração de um pacote de aplicativo Android, são necessárias informações que requerem confidencialidade, tais como a senha da chave do certificado usado para assinar o arquivo `apk`.

Tais informações não podem estar descritas explicitamente no arquivo `build.gradle` da aplicação. Porém, podem estar descritas de forma protegida no software de integração que construirá o projeto, no caso o Jenkins:

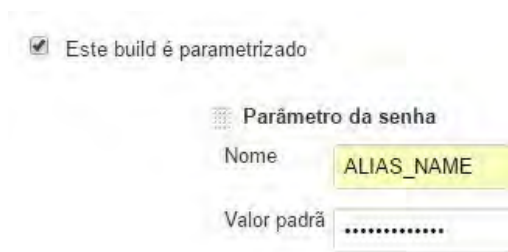


Figura 4.38: Configurando build parametrizado

Outro diferencial da configuração do job referente a este estágio está relacionado à preparação do emulador a ser disparado para a execução dos testes funcionais.

Ambiente de build

☐ Assign unique TCP ports to avoid collisions

☒ Run an Android emulator during build

☐ Run existing emulator

☒ Run emulator with properties

Android OS version

Screen density

Screen resolution

Device locale

 Locale will default to 'en_US' if not specified

SD card size

Target ABI

Emulator name suffix

Hardware

Common emulator options

☒ Reset emulator state at start-up

☐ Show emulator window

☐ Use emulator snapshots

Figura 4.39: Configurações do emulador

Na figura, são exibidas características do dispositivo emulado para a execução dos testes, como a versão do sistema operacional Android, densidade da tela e resolução da tela. É importante notar que a opção `Reset emulator state at start-up` está selecionada. Ela determina que, a cada início de execução do estágio `RachaContaAndroidInstrumented`, seja usado um emulador novo sem resíduos de execuções anteriores, de modo que os resultados dos testes funcionais sejam confiáveis.

Como mencionado anteriormente, o Android Emulator Plugin do Jenkins pode apresentar muitos problemas. Logo, é sugerido obter uma versão estável dele, de modo a viabilizar a construção do estágio `RachaContaAndroidInstrumented`. No [GitHub](https://github.com/jenkinsci/android-emulator-plugin) desse plugin (<https://github.com/jenkinsci/android-emulator-plugin/releases>) são disponibilizadas diversas versões. Uma versão que é considerada mais estável que outras é esta:



Figura 4.40: Versão estável do Android Emulator Plugin

Deve ser realizado o download do arquivo `android-emulator.hpi`. Então, pelo Jenkins, acesse o menu `Gerenciar Jenkins > Gerenciar plugins`. Em seguida, acesse a aba `Avançado` e, por fim, realize o upload do plugin através da seção `Atualizar plugin`.

Uma vez que o Jenkins seja reiniciado, o estágio de testes funcionais tende a ser executado sem problemas. Vale frisar: tende. Lembrando, trata-se de um plugin instável. Pode ser necessário algum esforço adicional para seu correto funcionamento.

O próximo estágio a ser configurado é responsável pelo

empacotamento do app *Racha Conta* na forma de um arquivo de aplicativo Android (ou seja, com extensão `.apk`), que se chama `RachaContaAndroidAssemble`.

A task Gradle, responsável pelo processo de empacotamento da aplicação, deve ser definida, como também a disponibilização do artefato resultante do processo de empacotamento, ou seja, do `apk` do aplicativo Android.



The image shows the 'Build' section of a Jenkins job configuration. It includes a 'Invoke Gradle script' section with radio buttons for 'Invoke Gradle', 'Use Gradle Wrapper' (selected), and 'Make gradlew executable' (checked). Below this are checkboxes for 'From Root Build Script Dir' (checked) and a text field for 'Build step description'. There is also a 'Switches' text field. At the bottom, the 'Tasks' field contains the text 'assembleRelease'.

Figura 4.41: Definindo task do processo de empacotamento



The image shows the 'Ações de pós-build' (Post-build Actions) section of a Jenkins job configuration. It features a section titled 'Arquivar os artefatos' (Archive the artifacts) with a sub-label 'Arquivos para arquivar' (Files to archive). The text field next to it contains the wildcard pattern '**/build/outputs/apk/**release*.apk'.

Figura 4.42: Configuração para disponibilização do arquivo do aplicativo Racha Conta

Uma vez que a execução do job *RachaContaAndroidAssemble* tenha sido finalizada com sucesso, o arquivo *apk* do *Racha Conta* estará disponível.

Projeto RachaContaAndroidAssemble

Estágio responsável pelo empacotamento da aplicação "Racha Conta".

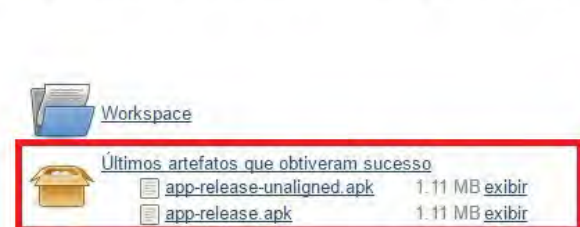


Figura 4.43: Disponibilização do arquivo do aplicativo Racha Conta

O último job a ser configurado é responsável pela publicação do aplicativo *Racha Conta* no Google Play. O job `RachaContaAndroidPublish` tem somente duas diferenças significativas em relação aos jobs anteriormente configurados.

A primeira é a necessidade de fornecimento do *service account* usado para publicação no Google Play. Um *service account* é um mecanismo que permite o uso da Google Play Developer Publishing API (que será melhor explicada em seções posteriores) através de uma aplicação em vez de um usuário comum. A aplicação, no contexto dessa seção, é o Jenkins. Assim como, em seções anteriores, a aplicação foi o Travis CI e o GoCD.

A informação sobre o *service account* é fornecida na seção de build parametrizado da configuração do job. Logo, o job `RachaContaAndroidPublish` executa um build parametrizado.

Parâmetro da senha

Nome: SERVICE_ACCOUNT_EMAIL

Valor padrão:

Descrição:

[HTML escapado] [Visualizar](#)

Figura 4.44: Fornecimento do service account

Por último, é necessária a configuração da task Gradle de publicação.

Build

Invoke Gradle script

☐ Invoke Gradle

☒ Use Gradle Wrapper

Make gradlew executable ☒

From Root Build Script Dir ☒

Build step description

Switches

Tasks

Figura 4.45: Configuração da task de publicação

Agora temos todos os estágios do pipeline finalmente configurados! Porém, apesar de sempre fazermos referências aos jobs como estágios, intuitivamente não temos até o momento o pipeline como se estivesse composto por estágios, pela falta de uma

representação visual do pipeline. Essa representação, contudo, será possível com o uso do *Build Pipeline Plugin*. Para a criação dessa representação visual, segue sua definição:

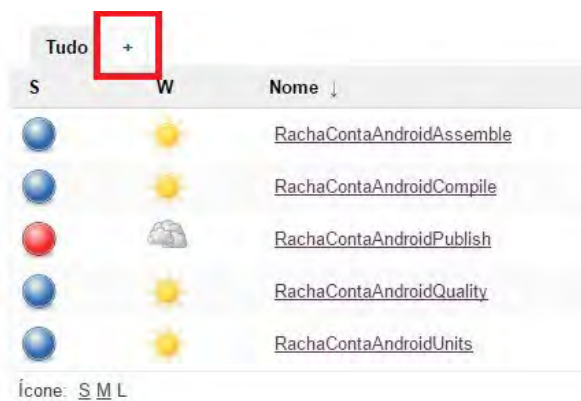


Figura 4.46: Estágios do pipeline do Racha Conta



Figura 4.47: Definindo a representação visual do pipeline

A configuração essencial do pipeline é simples. Basta determinar o job inicial. No caso do pipeline do *Racha Conta*, esse job é o *RachaContaAndroidCompile*. Tendo o pipeline configurado, o painel de visualização é exibido como a seguir:

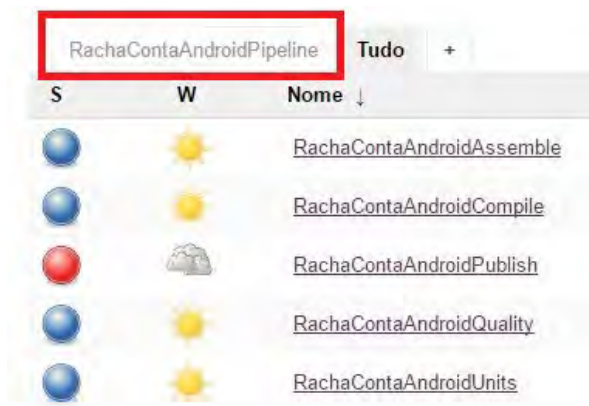


Figura 4.48: Opção de menu para o pipeline do Racha Conta



Figura 4.49: Representação visual do pipeline do Racha Conta (retângulos verdes e vermelhos)

É primordial que pipelines de deployment sejam visuais, pois eles fornecem feedback com muito eficácia. Por exemplo, cada retângulo na figura anterior representa um estágio. Retângulos de cor verde expressam que a execução do pipeline naquele estágio foi finalizada com sucesso. Já retângulos de cor vermelha expressam algum tipo de falha em seu processamento.

Como demonstrado no *capítulo 3*, esse pipeline construído pelo Jenkins poderia ser exibido em um monitor em uma sala ampla, por exemplo, de modo que todas as pessoas interessadas no atual estado do aplicativo saberiam imediatamente caso algo de errado estivesse

acontecendo durante o processamento do build.

Analisando individualmente a representação visual de um estágio do pipeline, temos:



Figura 4.50: Detalhes de um estágio de pipeline

1. Rótulo de execução do estágio
2. Nome do estágio
3. Data e horário da execução do estágio
4. Quantidade de tempo gasta para execução do estágio
5. Indica que o estágio requer parâmetros para sua execução
6. Saída do console contendo logs da execução do estágio
7. Botão para reexecução do estágio

Ainda, um cenário possível do pipeline é o seguinte:

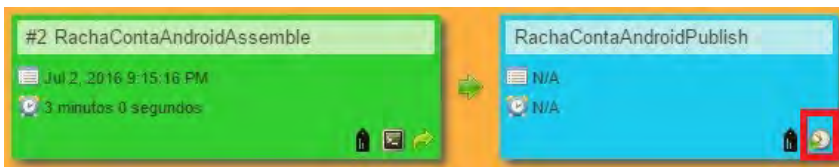


Figura 4.51: Trigger manual de um estágio

No botão sinalizado na figura, o estágio `RachaContaAndroidPublish` pode somente ser ativado manualmente. Esse contexto em específico é viável quando, por

exemplo, por estratégia da área comercial da empresa proprietária do aplicativo, prefere aguardar uma data ou ocasião para sua publicação. Isso é *continuous delivery*. Em caso de *continuous deployment*, 100% do pipeline seria automatizado, sem a necessidade de intervenção manual.

Para viabilizar esse cenário, o estágio RachaContaAndroidAssemble precisa ser reconfigurado, pois ele estava anteriormente preparado para ativar o estágio RachaContaAndroidPublish automaticamente. Logo, a configuração de *trigger* automático é excluída e, em seu lugar, é habilitada a configuração de *trigger* manual:



Figura 4.52: Configurando trigger manual do próximo estágio

4.4 COMPARAÇÃO ENTRE FERRAMENTAS

Comparações são sempre polêmicas. A melhor ferramenta dentre as apresentadas é... Depende! Desenvolver software profissionalmente de forma correta é difícil.

Complexidade é algo presente e constante no dia a dia de um time de desenvolvimento de software. Prazos, necessidades, custos, nível técnico dos membros de equipe, tudo deve ser levado em consideração na tomada de decisões. E a ferramenta usada para integração do software também merece ser avaliada

minuciosamente, segundo um conjunto de fatores.

Existem muitos elementos que podem ser levados em conta na escolha da ferramenta de integração ideal. A importância de tais elementos é subjetiva. A lista a seguir é resultante de pesquisa (sobre a documentação de cada ferramenta e opiniões de usuários), experimentos realizados e experiência acumulada do uso frequente delas. Tais fatores são descritos a seguir.

Containerização

Contêineres provêm ambientes que isolam a dependência de máquina para os softwares que são manipulados dentro deles. Eles também fazem uso de primitivas do sistema operacional onde são instalados, de modo que eles possam ser instanciados e destruídos rapidamente, em contrapartida a máquinas virtuais convencionais.

O **Travis CI** permite o build sobre aplicações em ambientes containerizados, de forma que a construção de aplicações sob esses ambientes não será afetada sobre problemas típicos em integrações, como: dependência de sistema operacional, máquinas trabalharem com sistemas de 32 ou 64 bits, dependências de softwares auxiliares para seus builds, dentre outras preocupações.

Já o **GoCD** e **Jenkins**, pelo fato serem *self-hosted*, poderão apresentar dependências. Contudo, viavelmente possíveis de serem tratadas.

Visibilidade do pipeline

Durante a execução de um build no **Travis CI**, é possível somente ver o log dessa execução em tempo real e se algo está errado. Ao final do build, também é possível saber se o build falhou ou passou. Porém, não é possível com essa ferramenta mostrar o pipeline com seus estágios dispostos, de modo a permitir a gestão

visual por partes interessadas no atual status da integração do aplicativo.

Já com as ferramentas **GoCD** e **Jenkins** (através do uso de plugins), essa disposição é possível por padrão. Isso é muito importante, já que feedback rápido é uma das características mais importantes em ferramentas de integração.

Complexidade de workflows

Jobs podendo ser executados em paralelo, com deploys sendo disparados em múltiplos ambientes e monitoramento de várias branches simultaneamente... Em cenários nem tão complexos como esse, o **Travis CI** pode não oferecer suporte total, sendo necessário o auxílio de um *shell script* para a customização do workflow.

Já o **GoCD** e o **Jenkins** são ferramentas poderosas em relação a gerenciamento de workflows complexos. Além de permitirem o suporte de um *shell script* no gerenciamento de seu pipeline, por vezes ele não se faz necessário.

Através de suas IDEs (principalmente do GoCD), essa complexidade de workflows pode ser gerenciada. Ademais, quando o Jenkins não oferece suporte suficiente, por default, basta a adição de plugins à ferramenta para que ela seja capaz de suprir tal tarefa.

Extensibilidade por plugins

O servidor de integração **Travis CI** não permite a instalação de plugins para que o possibilite ser mais extensível quanto a novas funcionalidades. Já o **GoCD** e, principalmente, o **Jenkins** têm isso como uma de suas principais características.

O GoCD (ao menos no momento em que este livro está sendo escrito) ainda oferece poucos plugins compatíveis que sejam úteis

para integração de aplicações Android. O principal deles é o *Gradle Plugin*. Já o **Jenkins** tem entre suas principais qualidades um amplo espectro de plugins disponíveis, inclusive para integração de apps Android.

É necessário reaproveitar os dados de um relatório para exibí-los visualmente? Existe um plugin que trata esse problema. Preciso de um plugin que notifique por SMS stakeholders quando um build quebra. Existe também um plugin para isso. Preciso de um plugin em que o Chuck Norris dá uma bronca no time de desenvolvimento quando o build quebra. Até para isso existe plugin, acredite. Ou seja, desde plugins essenciais até os não tão essenciais, permitem com que o Jenkins fique com a cara que seus responsáveis desejam e atenda seus problemas da forma mais conveniente.

Usabilidade da ferramenta

Preciso simplesmente integrar meu app Android do repositório de código-fonte ao Google Play. Essa é uma tarefa simples com o uso do **Travis CI**, pois, para um simples workflow como aquele descrito, a interface gráfica da ferramenta oferece uma experiência de usuário muito intuitiva. O mesmo vale para o **GoCD**.

Contudo, quando analisando a usabilidade do **Jenkins**, isso não vale. Apesar do poder dessa ferramenta, o seu operador precisa descobrir como extrair todo o seu potencial, ou experimentando-a, ou através de fontes na internet que explicam como resolver determinados problemas encontrados durante sua configuração.

Confiança na ferramenta

O que significa "confiança na ferramenta"? Por exemplo, no **Travis CI** e **GoCD**, uma vez que o usuário interaja diretamente com sua interface gráfica para o ajuste de determinada configuração, essa é imediatamente aplicada. A mesma experiência com o **Jenkins**,

uma boa quantidade de vezes, não é verdadeira, fazendo-se necessária a sua reinicialização para que a configuração seja aplicada.

O problema desse cenário são vários jobs já estarem configurados (com alguns em execução) e ser necessária a reinicialização da ferramenta devido a um específico job. Ou seja, o Jenkins não costuma ser tão confiável quanto as outras ferramentas, o que não significa que seja uma ferramenta ruim diante de suas tantas outras qualidades.

Expondo todos esses fatores relatados, visualmente temos:

	Travis CI	GoCD	Jenkins
Containerização	✓	✗	✗
Visibilidade do pipeline	✗	✓	✓
Complexidade de workflows	✗	✓	✓
Extensibilidade por plugins	✗	✓	✓
Usabilidade da ferramenta	✓	✓	✗
Confiança na ferramenta	✓	✓	✗

Figura 4.53: Comparação entre ferramentas

Apesar de não ser a melhor ferramenta avaliada de acordo com os prós e contras expostos na tabela, o Jenkins é muito mais maleável graças a seu arsenal de plugins. Em consequência disso, o que ele entrega de valor ao usuário é maior em relação às outras ferramentas.

Como profissionais da área de produção de software, é fundamental que tenhamos a naturalidade em realizar experimentos. *O Jenkins não resolve tal problema.* Pesquise por plugins que possam resolver esse problema (ou mesmo,

experimente desenvolvê-los do zero).

O Jenkins tem um bug quando certo plugin é aplicado. Vá ao GitHub onde está hospedado o Jenkins (<https://github.com/jenkinsci/jenkins>) e entre nas discussões de issues abertas para ele. Algumas respostas para problemas estarão expostas lá. Sendo um profissional de desenvolvimento de software ou responsável no setor de infraestrutura, é quase um requisito esse interesse constante em construir e desconstruir soluções. E isso é essencialmente o que o Jenkins permite através de seus plugins.

Já está disponível o Jenkins 2.0 para download. Essa versão do Jenkins permite com que um pipeline por completo seja descrito como código (*pipeline as code*) através de um arquivo com scripts de build, semelhante ao Travis CI. Pipelines expressos dessa forma permitem que, em um ambiente com cultura DevOps, tanto responsáveis pela área de desenvolvimento (Dev) quanto pela área de operação (Ops) comuniquem-se por uma linguagem compreensível a todos.

Além disso, outro ponto a favor é permitir compartilhar a configuração do pipeline como código por meio de um repositório compartilhado, tal como o GitHub. Apesar da possibilidade de codificar um pipeline inteiro como código, a versão 2.0 do Jenkins (no momento em que este livro está sendo escrito) apresenta ainda muitos bugs. Porém, conforme o passar do tempo, a tendência é esse cenário melhorar e ser recomendada sua adoção.

4.5 PUBLICAÇÃO NO GOOGLE PLAY

Todos os estágios que compõem um pipeline de integração contínua são igualmente importantes. Porém, um estágio merece especial atenção, pois, tendo sido determinado que uma aplicação será construída e validada de forma automatizada, outra pergunta cabe: *automatizar ou não o deploy?*

Não existe resposta correta para essa pergunta. A resposta mais apropriada é (mais uma vez) depende! Depende da necessidade do proprietário da aplicação. Além disso, automatizar a etapa de publicação é o que diferencia o processo de entrega em *continuous delivery* e *continuous deployment*. Ou seja, uma vez que o pipeline inicie sua execução, caso a aplicação tenha sido corretamente construída e validada, ela sempre será automaticamente publicada no Google Play (no caso de apps Android), ou dependerá da ação de alguém determinar se ela será publicada ou não.

Independente se o estágio de publicação será engatilhado automaticamente ou não, quem despachará o apk em direção ao Google Play será a ferramenta de automação, seja ela o Travis CI, GoCD ou Jenkins. E para que isso seja possível, elas farão uso da **Google Play Developer Publishing API**. Essa API serve como interface entre a ferramenta de automação e o Google Play:

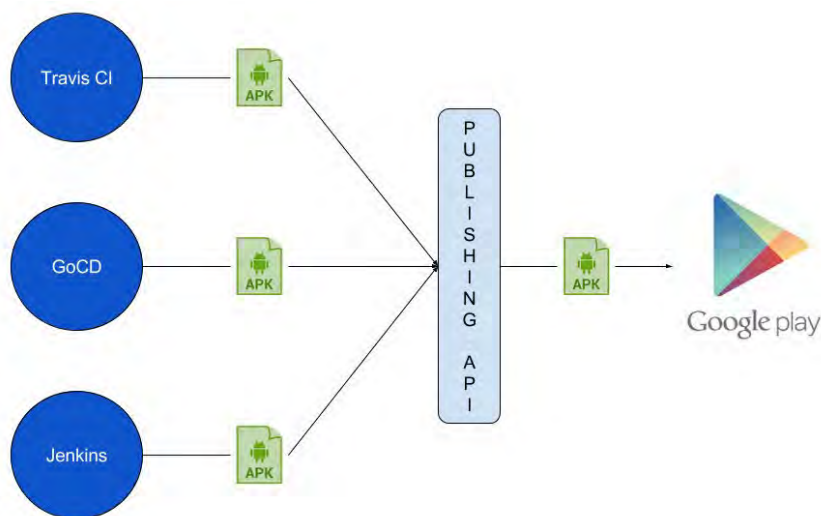


Figura 4.54: Publicação pela Google Play Developer Publishing API

O uso desta Web API requer o entendimento dessa através de sua documentação (<https://developers.google.com/android-publisher/>). Porém, existe um caminho muito mais simples. Trata-se do plugin *gradle-play-publisher*. Ele simplifica a publicação de um arquivo apk e seus metadados ao Google Play por meio de tasks Gradle, sem a necessidade da implementação de chamadas explícitas à Web API.

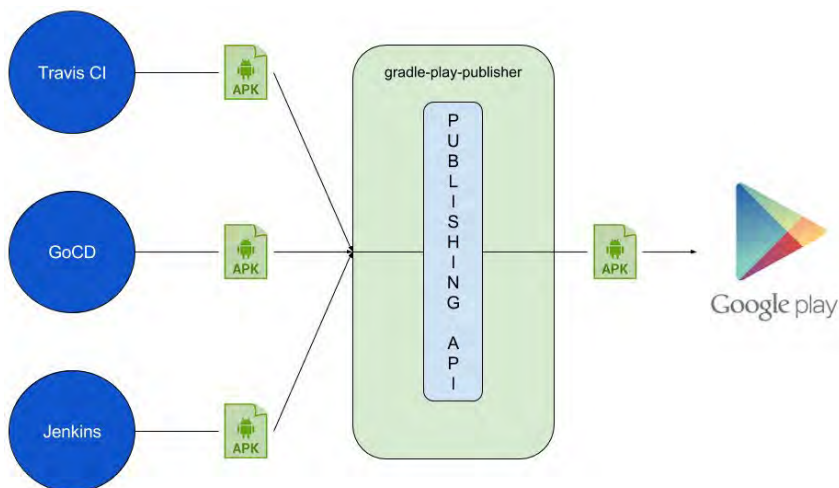


Figura 4.55: Publicação através do gradle-play-publisher

O GitHub do *gradle-play-publisher* é a principal fonte de informações sobre configuração e uso desse plugin: <https://github.com/Triple-T/gradle-play-publisher>.

Ou seja, como mostra a figura, o *gradle-play-publisher* é uma espécie de invólucro sobre a Google Play Developer Publishing API, tal que solicitações HTTP explícitas a essa Web API tornam-se desnecessárias.

Automatizando a publicação

Para viabilizar a automatização de um app Android no Google Play, o primeiro passo é publicá-lo manualmente. Sim, a primeira publicação deve ser manual, pois é necessário o registro do `applicationId` do aplicativo na plataforma de distribuição do Google, e isso não pode ser realizado através da Web API de publicação.

A publicação manual do aplicativo no Google Play requer que o arquivo apk seja gerado. Todo apk Android deve ser assinado por um certificado digital. Portanto, esse certificado deve ser gerado.

Para gerá-lo, selecione a opção de menu do Android Studio **Build > Generate Signed APK...** e preencha todas as informações solicitadas. Ao final do processo, um arquivo denominado keystore (com extensão .jks) será gerado e o arquivo apk referente ao aplicativo poderá ser criado.

Mais detalhes sobre o ato de assinar apps Android podem ser encontrados em <https://developer.android.com/studio/publish/app-signing.html>.

Ao determinar onde o apk deverá ser armazenado ao final do processo de sua geração, preferencialmente, escolha o seguinte local:

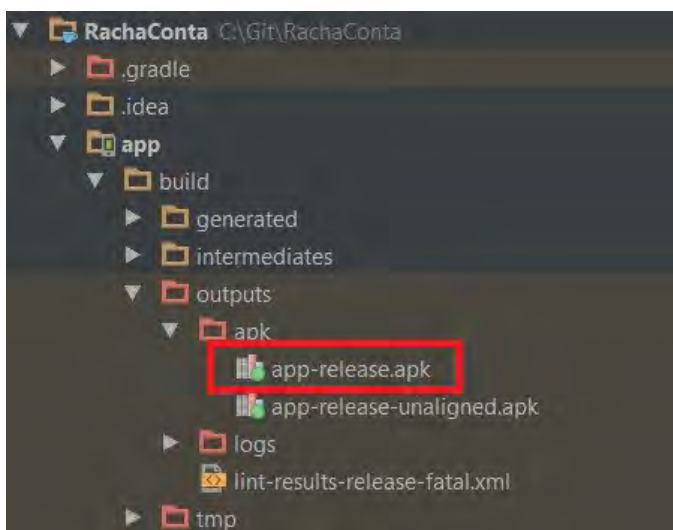


Figura 4.56: Localização de apk gerado

Também é possível gerar o mesmo apk através da execução da task Gradle `assembleRelease` no console do Android Studio. Porém, para que isso seja possível, as devidas configurações devem ser adicionadas ao arquivo `build.gradle` em nível de módulo:

```
signingConfigs {  
    release {  
        storeFile rootProject.file('rachaconta.jks')  
        storePassword System.getenv("KEYSTORE_PASSWORD")  
        keyAlias System.getenv("ALIAS_NAME")  
        keyPassword System.getenv("ALIAS_PASSWORD")  
    }  
}
```

Figura 4.57: Configurando apk release — 1

```
buildTypes {  
    release {  
        signingConfig signingConfigs.release  
        minifyEnabled false  
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'  
    }  
}
```

Figura 4.58: Configurando apk release — 2

O arquivo `keystore`, gerado pela opção de menu `Generate Signed APK...`, deve ser adicionado no folder raiz do projeto do app, para que possa ser indexado através da instrução `rootProject.file`. O restante das configurações faz uso da instrução `System.getenv`, que permite que valores sejam acessados de variáveis de ambiente do sistema.

No contexto de uma ferramenta de integração contínua como, por exemplo, o Jenkins, será possível configurar builds

parametrizados, em que os parâmetros do build (isto é, variáveis de ambiente) podem ser definidos, obtidos durante o processamento do build da aplicação e substituídos nas instruções contidas no arquivo `build.gradle`, que fazem referências a esses parâmetros. Isso permite que informações sigilosas, tais como senhas, não estejam explícitas no arquivo de configuração, e sim na ferramenta de integração, o que é mais conveniente quanto ao quesito segurança.

Além disso, vale mencionar a instrução `signingConfig` `signingConfigs.release`, que determina que `apk`s gerados com o tipo de build `release` sejam construídos com as informações fornecidas para esse tipo de build na seção `signingConfig`. Uma vez que as configurações para assinatura sejam todas fornecidas, executando a task Gradle `assembleRelease`, em caso de sucesso no build, fará com que o `apk` gerado seja armazenado no folder `apk`, como mostrado na figura anteriormente.

Uma vez com o `apk` do aplicativo disponível, é preciso acessar o painel do Google Play Developer Console. O painel pode ser acessado em: <https://play.google.com/apps/publish/>.

Uma informação importante: para publicar apps no Google Play, é necessário o pagamento de uma taxa única de US\$25,00.

Estando no painel de configuração, o botão `Add new application` deve ser pressionado e o nome do app preenchido. Por fim, deve ser selecionado o botão `Upload APK`, resultando na exibição do painel de administração para o app:

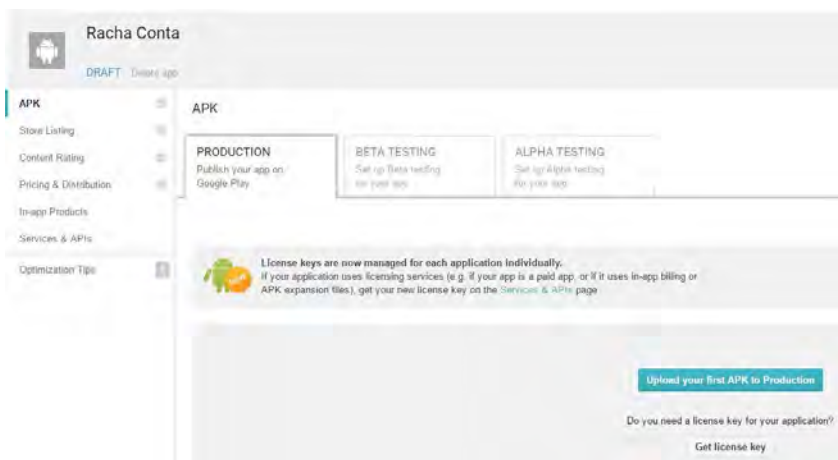


Figura 4.59: Racha Conta no Google Play Developer Console

Antes que o app seja publicado no Google Play, informações sobre ele terão de ser preenchidas em formulários contidos neste painel, como descrição do app a ser exibida no Google Play, classificação de faixa etária recomendada para o app, ícone do app, dentre outras informações. Uma vez que todas as informações sejam preenchidas, o botão `Upload your first APK to Production` deve ser pressionado.

Então, será realizado o upload do arquivo `apk` do aplicativo *Racha Conta* gerado através do Android Studio. E o app está oficialmente publicado no Google Play! Daqui em diante, todas as futuras publicações serão automatizadas. Assim, vamos configurar o processo.

Para permitir publicações de apps por ferramentas, em vez de seres humanos, é necessária a geração do **service account**. Ele permite com que servidores automatizados, tais como o Jenkins, façam uso da Google Play Developer Publishing API para a publicação de apps no Google Play. Para ter acesso à seção para a geração de um *service account*, no Google Play Developer Console,

selecione a opção de menu **Settings > API access** .

Informações mais detalhadas sobre como gerar um *service account* podem ser consultadas em: https://developers.google.com/android-publisher/getting_started.

Uma vez finalizado o processo de geração, será disponibilizado o e-mail do *service account* e um arquivo com extensão **.p12** , que se trata de uma chave.



Figura 4.60: E-mail do service account

Já a chave p12 deve ser colocada na raiz do projeto do aplicativo *Racha Conta*.

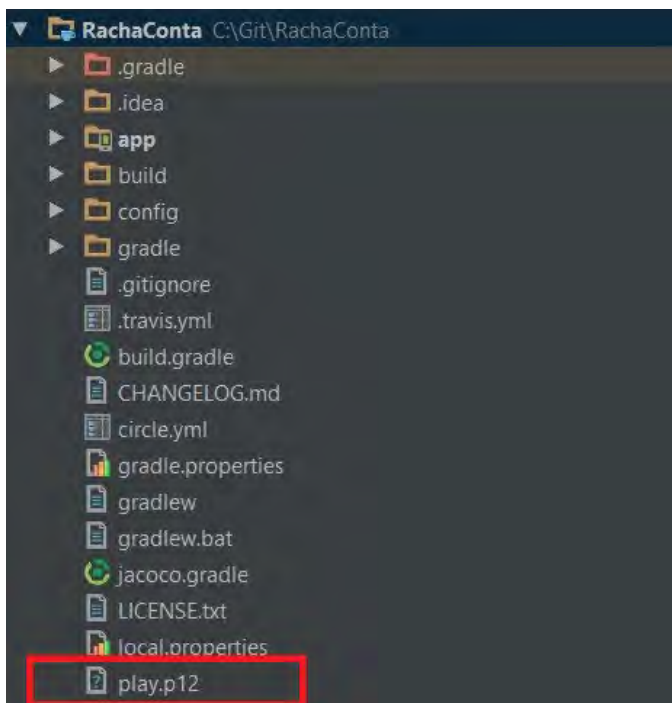


Figura 4.61: Chave p12

Outra configuração a ser definida é o conjunto de permissões que a ferramenta automatizada estará habilitada a exercer sobre a API de publicação. O conjunto mínimo de permissões para viabilizar a publicação de um app no Google Play deve ser setada desta forma:



Figura 4.62: Permissões para o service account

Prosseguindo com as configurações, chega-se à etapa de obtenção e aplicação do plugin *gradle-play-publisher* ao projeto do aplicativo. No arquivo `build.gradle`, em nível de projeto, o plugin deve ser obtido através da seguinte configuração:

```
buildscript {  
  
    repositories {  
        jcenter()  
    }  
  
    dependencies {  
        classpath 'com.android.tools.build:gradle:2.1.0'  
        classpath 'com.github.triplet.gradle:play-publisher:1.1.4'  
    }  
}
```

Figura 4.63: Obtenção do plugin `gradle-play-publisher`

Em seguida, esse plugin deve ser aplicado ao projeto no arquivo `build.gradle`, em nível de módulo.

```
apply plugin: 'com.android.application'  
apply plugin: 'com.github.triplet.play'
```

Figura 4.64: Aplicação do plugin `gradle-play-publisher`

Nesse mesmo arquivo `build.gradle`, deve ser configurado o bloco `play`. Esse bloco deve conter o e-mail do *service account* e a referência para a chave `p12`.

```
play {  
  
    serviceAccountEmail = System.getenv("SERVICE_ACCOUNT_EMAIL")  
    pk12File = rootProject.file('play.p12')  
}
```

Figura 4.65: Configuração do bloco `play`

Como mostrado na figura anterior, o e-mail do *service account* estará parametrizado na configuração do build da ferramenta de integração, pois esse trata-se de um dado sigiloso que é recomendado não estar literalmente exposto no `build.gradle`.

Por fim, devem ser configurados os metadados referentes ao app para exibição no Google Play. Metadados estes que tratam de informações exibidas ao usuário do app quando este estiver interessado em mais informações sobre o aplicativo, como a descrição do app, imagens de telas do app em execução, vídeo, e-mail de contato etc.

Os metadados devem ser dispostos sobre folders específicos dentro do projeto do aplicativo. A seguir, é mostrado que as últimas alterações sobre o app devem ser descritas no arquivo `whatsnew` no local especificado:

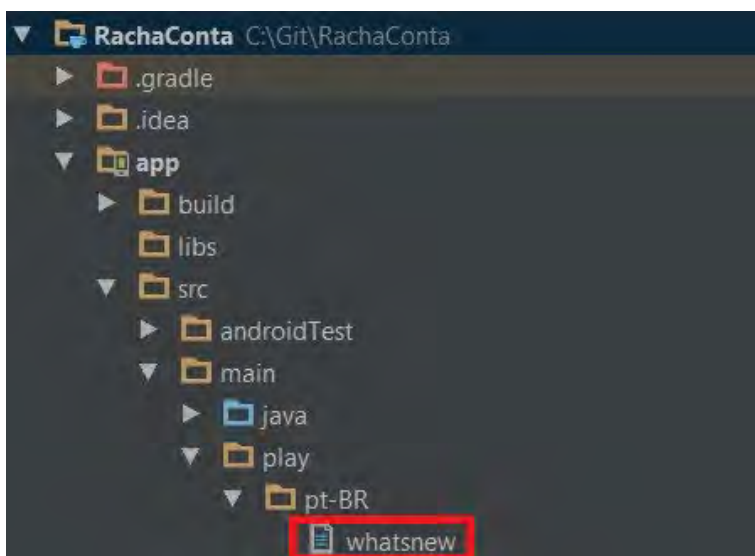


Figura 4.66: Recentes alterações sobre o app

Os folders de armazenamento de outros metadados podem ser consultados no GitHub do plugin *gradle-play-publisher*. E é isso!

Sim, todas as configurações para publicação automatizada de um app Android no Google Play foram definidas.

Para o próximo build sobre o app, o plugin *gradle-play-publisher* criará um conjunto de tasks Gradle. A mais importante delas é a task `publishApkRelease`. Ela realiza o upload do apk e recentes alterações sobre o app (contidas no arquivo `whatsnew`) para o Google Play. Essa task, como já mostrado anteriormente, é invocada pelas ferramentas de integração, tal que seja possível a elas publicarem um app ao final de um pipeline de integração contínua.

Criei um projeto de app Android do zero, implementei uma série de testes unitários, de integração, algumas regras de análise estática de código, testes funcionais, subi todos eles para o GitHub, configurei a ferramenta de integração para monitorar a estabilidade e validação do meu app e, então, publiquei-o para o Google Play através de um pipeline de entrega contínua. E agora?

Opa! Meus parabéns! Se você chegou até aqui, tenha em mente o seguinte: você agregou valor a seu app. Custo menor; confiabilidade sobre o app; maior proteção contra regressões; feedback rápido em caso de falha na integração; tarefas repetitivas automatizadas e tarefas de criação para o desenvolvedor.

O Google Play é uma possível alternativa para publicação de apps Android. Existem outras. O próximo capítulo tratará disso. São os mecanismos de distribuição denominados *over-the-air*.

DISTRIBUIÇÕES OVER-THE-AIR

Imagine uma pessoa diante de uma espécie de painel de operações em um website. Nesse painel, em um submenu, está disponível a opção `Instalar aplicativo`. Ao selecionar essa opção, o operador se depara com a possibilidade de realizar o upload de um arquivo correspondente a um app mobile, uma lista de nomes de dispositivos (smartphones e tablets) e um botão `Instalar`.

O operador realiza o upload do app, um dispositivo em especial é selecionado — o do próprio operador — e é pressionado o botão `Instalar`. E a seguinte mensagem é exibida no painel: *App instalado no(s) dispositivo(s) com sucesso*. Então, o operador tira seu smartphone do bolso e, como um "passe de mágica", o novo app está instalado e disponível para uso no *launcher* do dispositivo.

O operador era eu mesmo em 2012. Até então, com o mercado de mobilidade recém-despontando. Não tinha noção de que algo assim era possível de ser feito. Hoje, muitos ainda não conhecem sobre essa possibilidade.

5.1 O CONCEITO

Uma **distribuição over-the-air** (OTA) é um mecanismo de instalação (ou atualização) de sistemas (ou apps) sobre rede sem fio.

Ou seja, não é necessária a conexão do dispositivo mobile com um *host* via cabo para a transferência do software.

Atualizações do sistema operacional Android em dispositivos podem ser categorizadas como *over-the-air*, assim como atualizações de apps. Elas podem ser instalados/atualizados via *over-the-air* de diferentes formas. A primeira é realizada pelo despacho de um app através do site do Google Play, em que o usuário deve estar autenticado com sua conta do Google e um dispositivo mobile associado a ela.

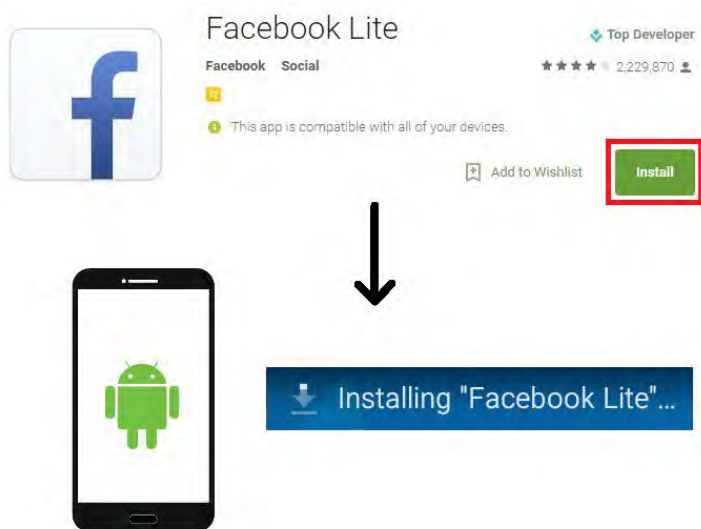


Figura 5.1: Over-the-air — 1

Outra categoria de distribuição *over-the-air* faz uso de *polling*. Através desse mecanismo, uma entidade faz consultas periódicas a outra entidade para verificar uma possível alteração de um status de interesse.

Um app instalado em um dispositivo Android pode estar associado a um serviço sendo executado em background que realiza consultas a um status contido em um servidor. Nesse caso, o status

de interesse seria o número da versão do app, de modo que, uma vez que a versão tenha sido incrementada e, logo, o app atualizado, uma distribuição *over-the-air* é ativada e o app atualizado enviado ao dispositivo para que seja, em seguida, instalado nele.

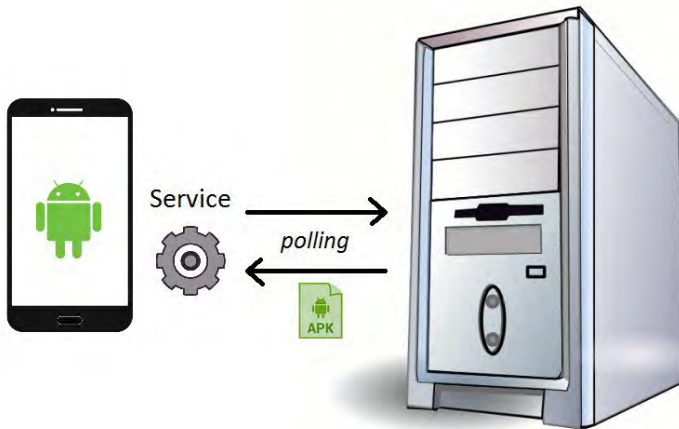


Figura 5.2: Over-the-air — 2

Um meio pelo qual também é possível realizar distribuições *over-the-air* se dá por ferramentas que, além de conter essa funcionalidade de distribuição, contém um aparato para monitoramento de tráfego de informações sobre apps. O mecanismo dessas ferramentas responsável pela distribuição de apps é conhecido como módulo de **distribuição beta**. As ferramentas permitem que um operador despache um novo app, ou sua atualização, para um dispositivo ou um grupo de dispositivos previamente selecionado.

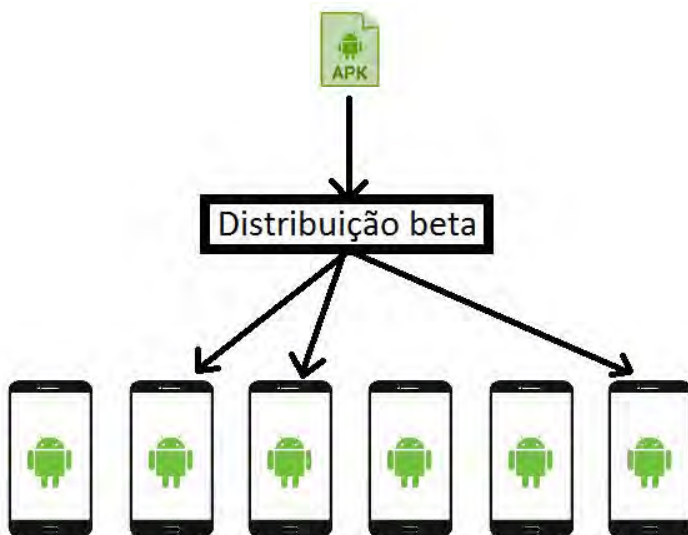


Figura 5.3: Over-the-air — 3

Apesar de ser beta em seu nome, não necessariamente a versão do app distribuído necessita ser beta. O nome vem do conceito de o app ser enviado a um conjunto de testadores em um ambiente controlado. Porém, é permitido distribuir apps em um ambiente controlado de usuários finais. Ainda neste capítulo, serão tratadas duas ferramentas que dispõem dessas funcionalidades: o HockeyApp e o Crashlytics.

5.2 REQUISITOS PARA ATUALIZAÇÕES OTA

Uma vez que a distribuição via *over-the-air* de um app é realizada, deve-se ter em mente um aspecto fundamental: arquivos do dispositivo-alvo serão modificados. Logo, o mecanismo pelo qual será efetuada essa distribuição deve, preferencialmente, atender ao conjunto de requisitos. Isso de modo que, após a instalação do app,

não faça com que o dispositivo demonstre inconsistência de dados e, por consequência, apresente falhas como, por exemplo, não realizar mais *boot*; realizar *reboot* infinitamente; e o app atualizado não iniciar sua execução corretamente.

Assim, de forma a prevenir tais cenários de falhas e também de modo a serem escaláveis e performáticos, mecanismos de distribuição *over-the-air* devem apresentar os seguintes requisitos:

- **Atualizações atômicas:** devem ser consistentes. Ou seja, ou a atualização é realizada por completo, ou o sistema-alvo permanece inalterado, como se nenhum comando para atualização houvesse sido realizado.
- **Tratamento sobre conexão ruim e falhas na transmissão:** uma vez que o processo de atualização tenha sido interrompido, somente as partes faltantes devem ser buscadas quando o download for retomado.
- **Flexíveis e reusáveis:** deve ser simples adicionar novos dispositivos-alvo para receber atualizações sem a necessidade de grandes ajustes na configuração do software de distribuição.
- **Processamento das atualizações em background:** o usuário do dispositivo pode operar normalmente sobre apps em foreground enquanto a atualização é realizada.

5.3 HOCKEYAPP

Esta ferramenta hospedada na nuvem permite a distribuição beta de apps a um conjunto de usuários previamente selecionados. Porém, não somente isso. Ela conglera um ambiente para coleta de *crash reports* em tempo real.

Mas o que isso significa? Digamos que um usuário do app, obtido via HockeyApp, enquanto esteja interagindo com o app em seu smartphone, ao pressionar um botão se depara com o seguinte alerta:

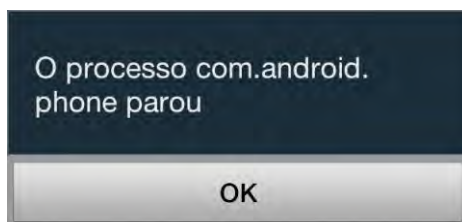


Figura 5.4: O app parou

Imediatamente, serão disparados para o HockeyApp detalhes do stacktrace da aplicação que vão auxiliar o desenvolvedor a compreender a causa do bug encontrado. Além disso, é fornecida pela solução uma porção de métricas que auxiliam no monitoramento do ciclo de vida da aplicação, como número total de usuários da aplicação, número total de crashes, número de novos usuários por dia, dentre outras.



Figura 5.5: Métricas de usuário — HockeyApp

A solução HockeyApp está disponível em <https://www.hockeyapp.net>.

Distribuindo apps via HockeyApp (sem CI)

Um app pode ser distribuído via HockeyApp para apenas um usuário. Porém, é comum a prática de criação de times de usuários, tal que, uma vez que uma nova versão de um app Android esteja disponível, seja possível a escolha de qual time de usuários receberá essa nova versão. E é essa categoria de distribuição que será abordada tendo em vista que se trata de uma prática muito comum na indústria de apps mobile.

Será demonstrada uma distribuição sem CI (*Continuous Integration*) primeiramente, para, na seção posterior, o mesmo mecanismo ser tratado através do estágio de publicação de um pipeline automatizado de CI.

Em primeiro lugar, deve ser criada uma conta de administrador no HockeyApp. Então, será acessível o painel de gerenciamento de apps. Através dele, será registrado um novo app.

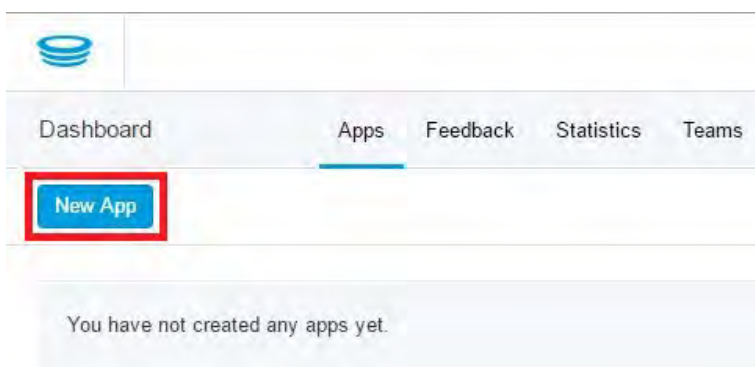


Figura 5.6: Distribuindo app via HockeyApp (etapa 1)

Será requerido que seja realizado o upload de um arquivo apk de release (no *capítulo 4*, há um passo a passo de como gerá-lo). Então, será solicitado que seja dada a autorização para que o app esteja disponível para download.

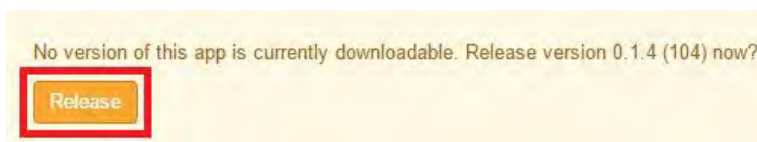


Figura 5.7: Distribuindo app via HockeyApp (etapa 2)

Assim, o app estará devidamente registrado, e todas atividades realizadas sobre ele por usuários devidamente monitoradas.

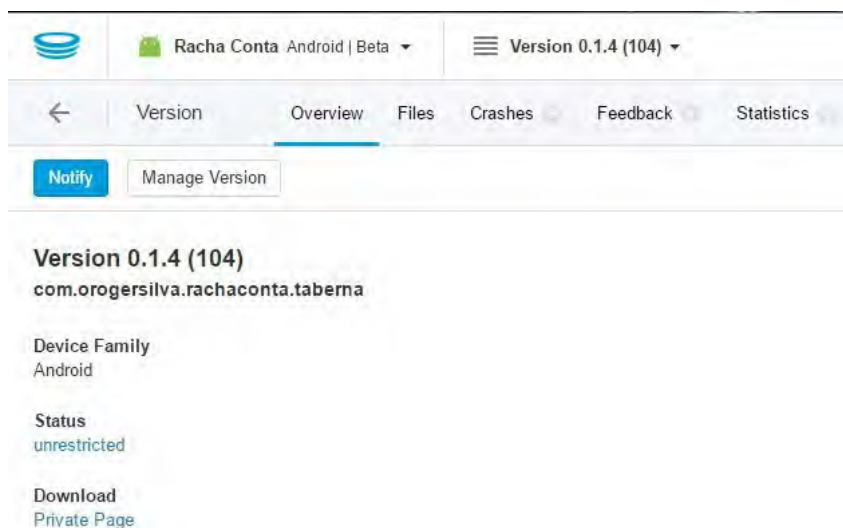


Figura 5.8: Distribuindo app via HockeyApp (etapa 3)

Em seguida, deve ser criado um novo time de usuários.

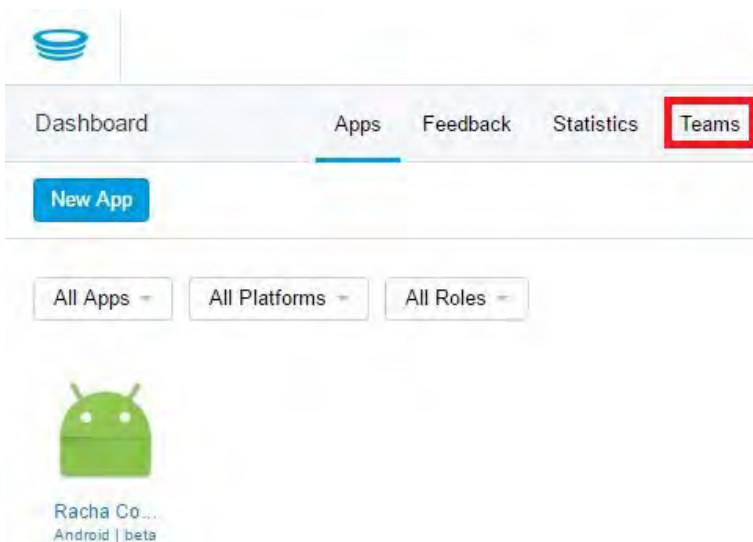


Figura 5.9: Distribuindo app via HockeyApp (etapa 4)

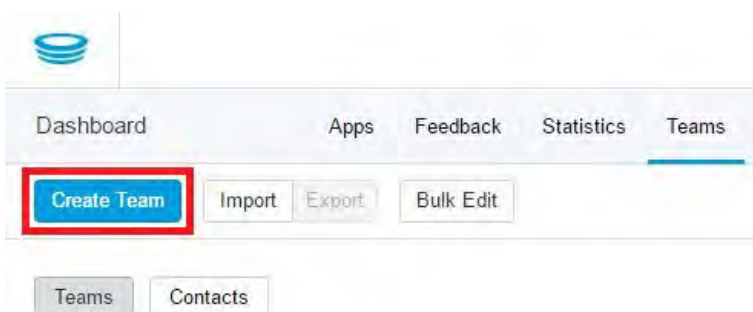


Figura 5.10: Distribuindo app via HockeyApp (etapa 5)

Ao ser criado o time, já é possível a adição de novos membros a ele.

Create New Team

Name

Racha Conta Testers

The name will be visible to all team members.

Tags

Enter one or more tags for this team. Tags will be applied to new users and apps.

Add Users

You can add or remove users now or later.

Email

+

john.appleseed@icloud.com

Press the enter key on your keyboard to add more users and edit the roles.

Email

✕

testeparahockeyapp@gmail.com

Invitation Email

☐ All Users

☒ Only New Users

Figura 5.11: Distribuindo app via HockeyApp (etapa 6)

Uma vez com o time criado, ele deve ser registrado na lista de membros a serem notificados quando uma nova versão do app estiver disponível para download.

5.3 HOCKEYAPP 149

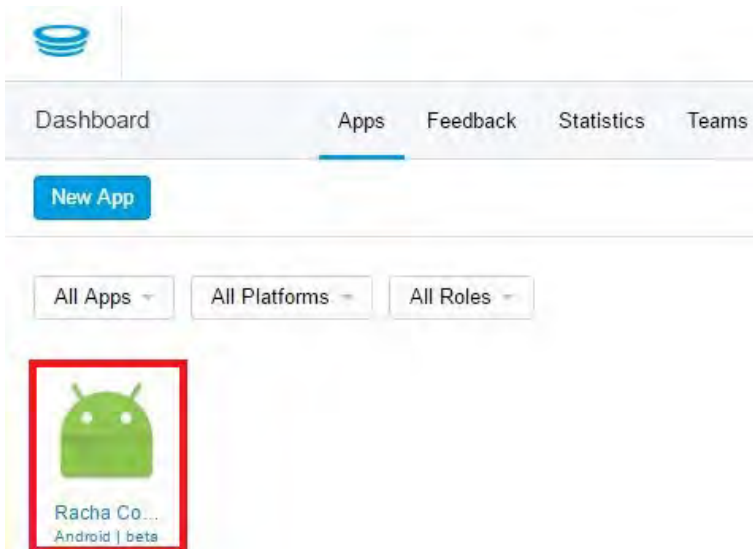


Figura 5.12: Distribuindo app via HockeyApp (etapa 7)

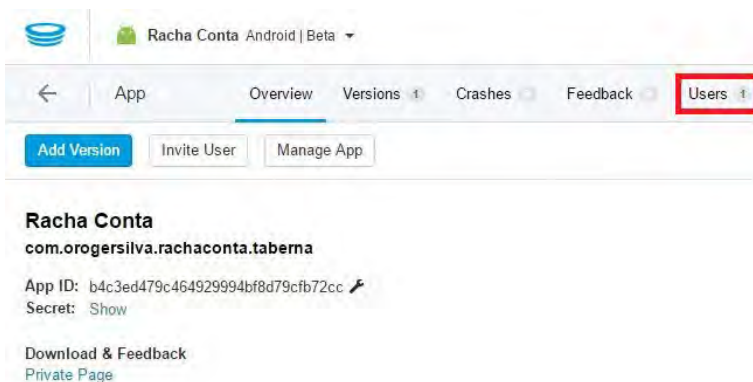


Figura 5.13: Distribuindo app via HockeyApp (etapa 8)

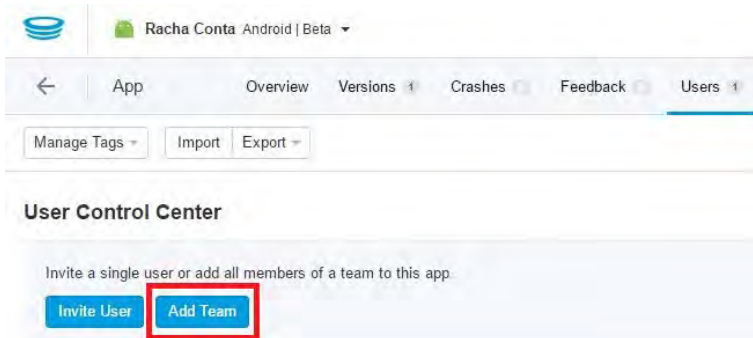


Figura 5.14: Distribuindo app via HockeyApp (etapa 9)

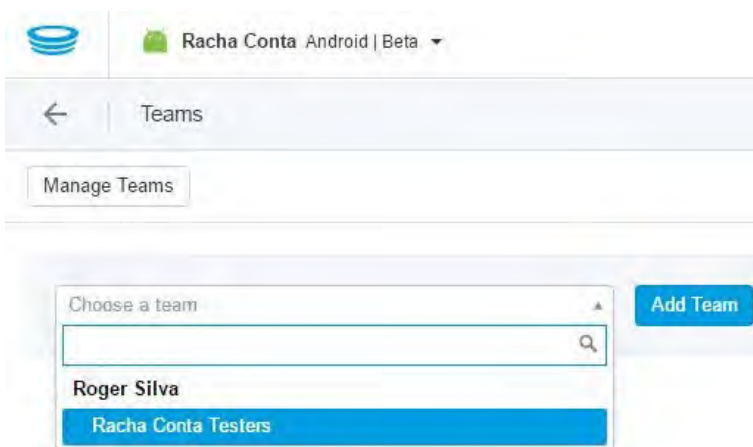


Figura 5.15: Distribuindo app via HockeyApp (etapa 10)

Add New Team

Team

Racha Conta Testers

Owner

Roger Silva

Users

1 Developer

0 Members

1 Tester

☒ Send Notification Email

A notification email will only be sent if you enable the checkbox above. This notification email is for informational purposes only and team members do not need to accept another invite.

Subject

Convite para Beta Tester

Message

Você foi convidado para ser beta tester do app Racha Conta.

Figura 5.16: Distribuindo app via HockeyApp (etapa 11)

Quando o app é atualizado, usuários serão notificados por e-mail sobre a nova atualização. Porém, na primeira notificação, será solicitado ao usuário para que ele cadastre-se no HockeyApp e também o dispositivo usado por ele para uso do app.

Racha Conta Invitation for HockeyApp

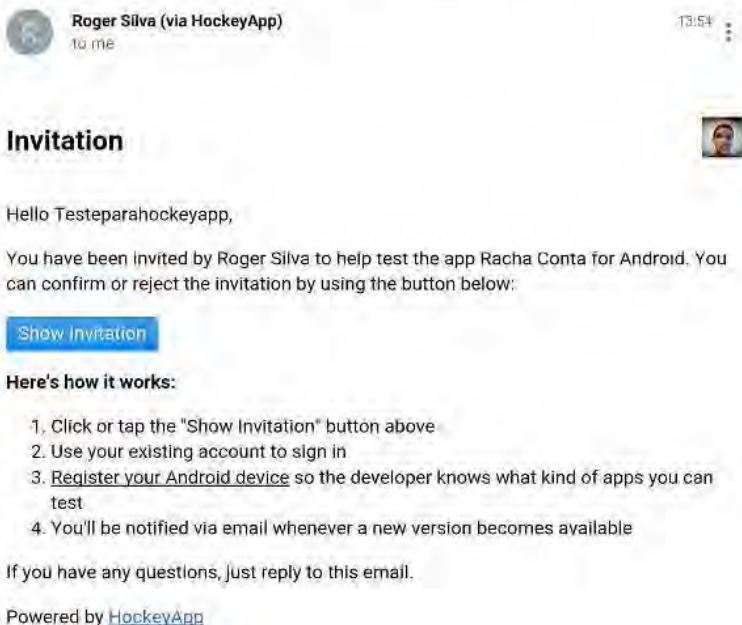


Figura 5.17: Distribuindo app via HockeyApp (etapa 12)

Estando o dispositivo registrado, o usuário terá acesso ao app Android do HockeyApp, pois esse será o meio pelo qual esse usuário terá acesso a novas versões distribuídas de apps de seu interesse.

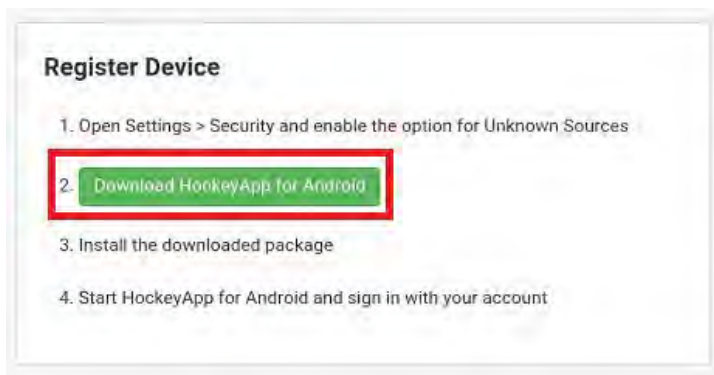


Figura 5.18: Distribuindo app via HockeyApp (etapa 13)

Por fim, o usuário pode acessar novas atualizações a cada vez que elas estiverem disponibilizadas, via o app HockeyApp.

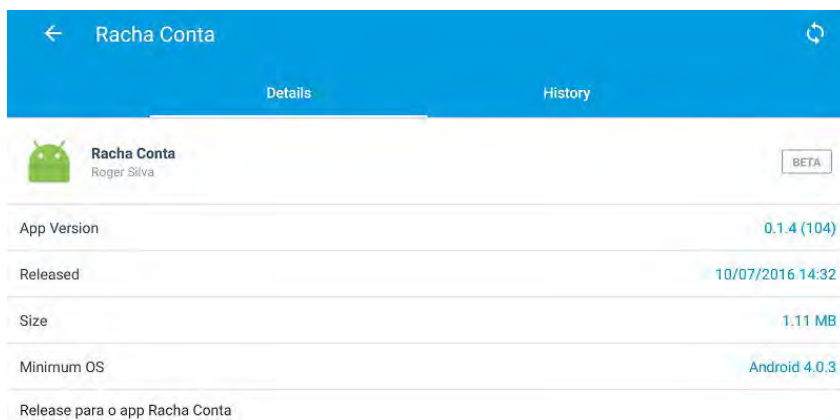


Figura 5.19: Distribuindo app via HockeyApp (etapa 14)

Distribuindo apps via HockeyApp (com CI)

Seria preciso a geração e publicação manual dessa atualização toda vez que fosse necessário distribuir uma nova atualização de app sem um mecanismo automatizado. É um processo tedioso e sujeito a erros.

Mas agora é com diversão! É possível configurar o estágio de publicação em um servidor de CI para que esse despache uma nova atualização de app Android para usuários via HockeyApp.

Para viabilizar esse mecanismo automatizado, será usado o *gradle-hockeyapp-plugin*. Trata-se de um plugin open source que facilita a implantação desse mecanismo.

Mais informações sobre o plugin podem ser obtidas em <https://github.com/x2on/gradle-hockeyapp-plugin>.

O plugin é configurado pelos arquivos `build.gradle` em níveis de módulo e projeto do app. No `build.gradle`, em nível de projeto, deve ser adicionada a seguinte dependência:

```
classpath 'de.felixschulze.gradle:gradle-hockeyapp-plugin:3.4'
```

Já no `build.gradle`, em nível de módulo, deve ser aplicado o plugin e definidas suas configurações.

```
apply plugin: 'de.felixschulze.gradle.hockeyapp'

hockeyapp {
    apiToken = System.getenv("HOCKEY_APP_API_TOKEN")
    notify = 1
    teams = 72599
}
```

O atributo `apiToken` pode ser obtido a partir da seção de configurações de conta do HockeyApp.

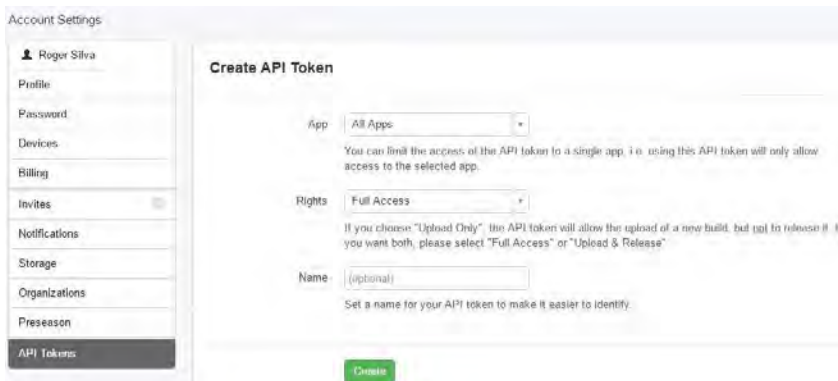


Figura 5.20: Obtendo o token de API

Deve ser concedido direitos de upload e release ao token.

Voltando ao `build.gradle`, o atributo `notify` permite com que, sempre que haja uma nova atualização do app tratado, o usuário seja notificado sobre isso. Já o atributo `teams` especifica para quais times deve ser distribuída a nova atualização. O identificador dos times pode ser obtido através da URL (mostrada na barra de endereços do navegador) quando aberto o painel de informações sobre esse time.

Figura 5.21: Identificador do time de usuários

Agora é necessária a configuração do estágio de publicação no servidor de CI. Ele será configurado no Jenkins. Porém, poderia ser configurado em quaisquer das ferramentas de integração contínua tratadas neste livro.

O job associado ao estágio levará o nome de `RachaContaAndroidPublishHockeyApp`. Muitas das configurações do job são as mesmas usadas pelo job `RachaContaAndroidPublish`, tratado no capítulo anterior, mas

com alguns detalhes adicionais. Na seção sobre parametrização do build, deve ser definido o token de API do HockeyApp.

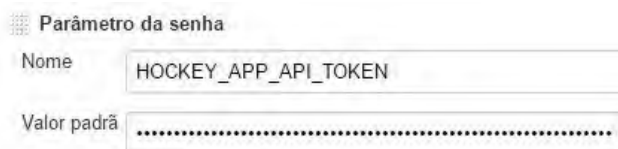


Figura 5.22: Configurando o token de API no Jenkins

Por fim, deve ser configurada a task `uploadReleaseToHockeyApp`, responsável pela publicação/distribuição do app via HockeyApp.



Figura 5.23: Configuração da task de publicação do HockeyApp

Sendo assim, quando disparada a execução do pipeline de deployment, todos os usuários pertencentes ao time definido no arquivo `build.gradle` serão notificados da existência de uma nova atualização do app. Isso faz do processo de distribuição menos tedioso e mais confiável, pela falta de necessidade de intervenção humana.

5.4 CRASHLYTICS

Assim como o HockeyApp, esta plataforma é hospedada na nuvem e trata-se de um kit de soluções de crash reporting, analytics e de distribuição de apps com enfoque em plataformas móveis. Ele faz parte de um conjunto maior de ferramentas denominado Fabric, pertencente ao Twitter.

Porém, apesar da diversidade de soluções fornecidas através do Crashlytics, estamos interessados, nesta publicação, em saber como entregar software Android da melhor forma possível. E a solução do kit que permite que isso seja possível tem o nome de **Beta by Crashlytics**.

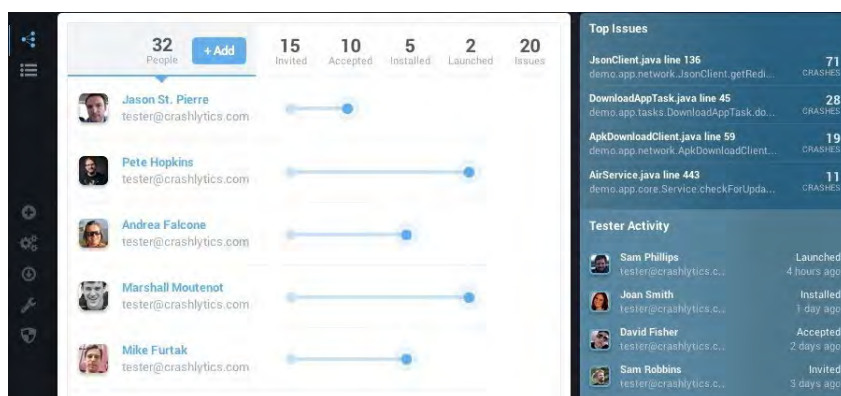


Figura 5.24: Dashboard do Beta by Crashlytics

A figura anterior retrata o dashboard de monitoramento de atividades sobre um app específico. Uma vez que um app Android é distribuído a usuários, todas as atividades relevantes ocorridas sobre o app serão expostas sobre esse dashboard. Uma atividade de interesse é, uma vez que um app tenha sido distribuído, como saber se esse app já foi instalado por algum de seus usuários?

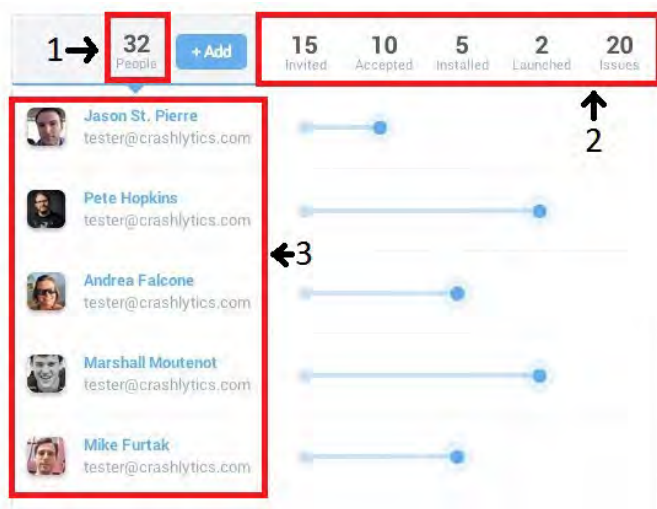


Figura 5.25: Dissecando o Beta by Crashlytics — 1

O subpainel anterior exhibe não somente a resposta para a pergunta anterior como também informações adicionais, tais como:

1. Total de usuários habilitados a receberem distribuições do app.
2. Total de convites enviados, total de convites aceitos, total de usuários que instalaram o app, total de usuários que já utilizaram o app e total de issues detectadas no app, respectivamente.
3. Lista dos usuários habilitados a receberem distribuições do app.

Outro subpainel do dashboard exhibe não somente crashes detectados sobre o app, como também a classe e a linha no código-fonte em que um bug foi manifestado.

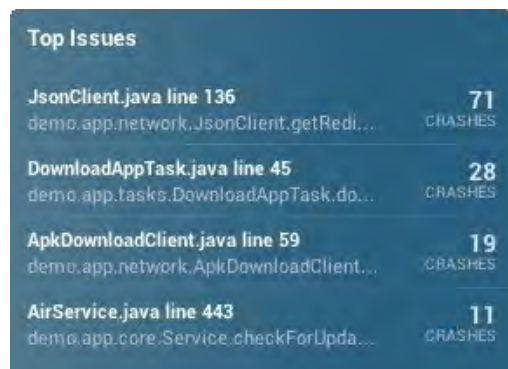


Figura 5.26: Dissecando o Beta by Crashlytics — 2

O último subpainel fornece informações sobre as últimas distribuições recebidas e operadas por usuários do app de interesse.



Figura 5.27: Dissecando o Beta by Crashlytics — 3

O kit de soluções Crashlytics pode ser acessado em <http://try.crashlytics.com/>.

Distribuindo apps via Crashlytics (sem CI)

Para distribuir um app Android via Crashlytics, primeiramente

é necessário o cadastro de uma conta de administrador no site da solução.

A seguir, acessando a opção de menu `File > Settings`, deve ser selecionada a seção `Plugins`. Depois, o botão `Browse repositories...` deve ser pressionado. Então, deve ser feita a busca por "Fabric for Android Studio", para que o plugin do Fabric seja integrado ao Android Studio.

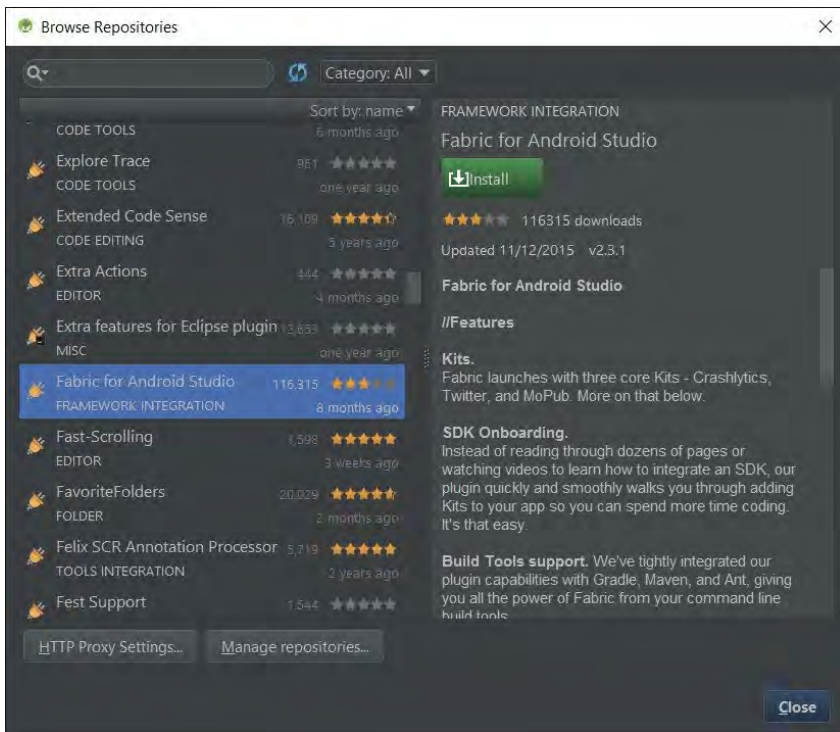


Figura 5.28: Instalando plugin do Fabric

Será solicitado que o Android Studio seja reiniciado. Em seguida, o Crashlytics será integrado ao projeto do app. Porém, em vez da necessidade de definir configurações nos arquivos `build.gradle` manualmente, o plugin do Fabric facilita (e muito) essa tarefa, pois ele analisa cada arquivo que deverá sofrer alterações

para adição do Crashlytics e, automaticamente, adiciona as instruções necessárias aos arquivos. Para isso, o plugin do Fabric deve ser acessado, ou pela barra de menu do Android Studio, ou de seu menu lateral.



Figura 5.29: Acesso ao plugin do Fabric

Uma vez acessado, basta confirmar a adição proposta pelo plugin.

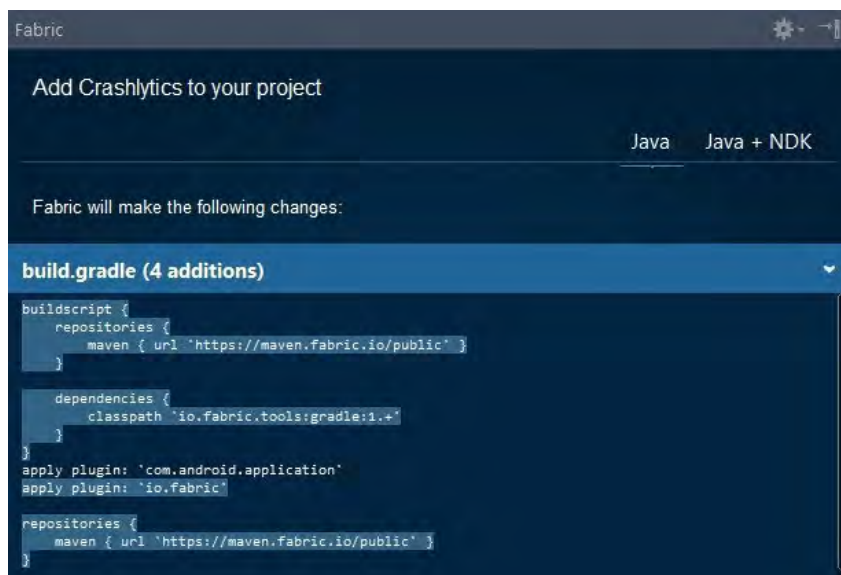


Figura 5.30: Adição de configuração para o Crashlytics

E o projeto está preparado para distribuição a usuários via

Crashlytics. Para essa distribuição ocorrer, o app de interesse deverá ser executado, no mínimo, uma vez para que seja registrado no Crashlytics e, então, arrastado sobre o painel disponibilizado pelo plugin do Fabric.

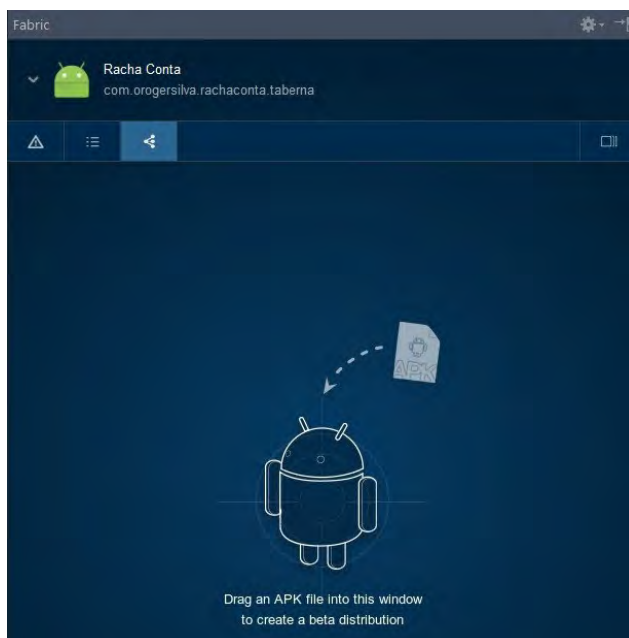


Figura 5.31: Distribuindo app via Crashlytics

E para quais usuários deverá ser distribuída a nova versão do app? É o que a próxima tela do painel do plugin solicitará. E isso se dá através do fornecimento do e-mail desses usuários.

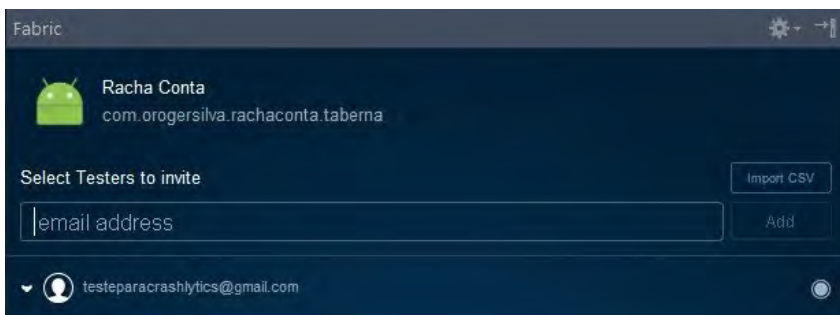


Figura 5.32: Definindo usuários de app no plugin do Fabric

Após a confirmação dos usuários, o app será distribuído a eles. E cada usuário receberá o convite pra uso desse app em sua caixa de e-mail.

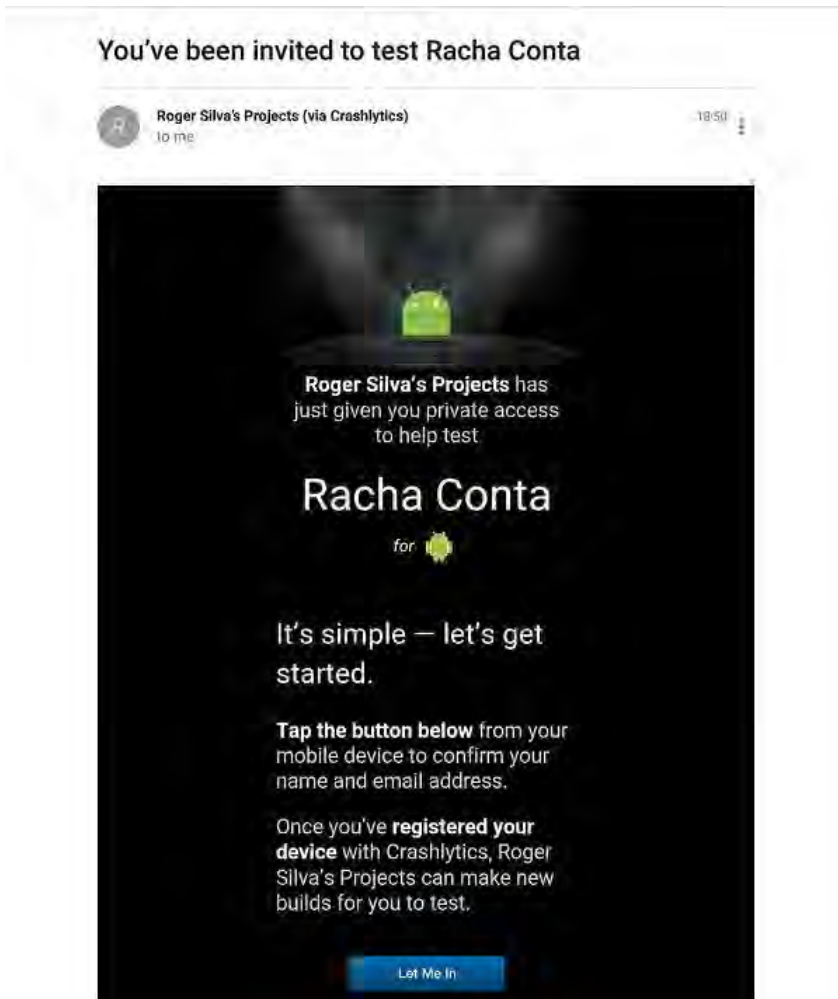


Figura 5.33: Recebimento do convite para uso do app via Crashlytics

Através desse e-mail, o usuário estará habilitado a realizar o download do app Android "Beta by Crashlytics", pois é por ele que todos os apps distribuídos via Crashlytics estarão disponíveis aos seus usuários. Logo, pelo e-mail, usuários serão direcionados ao navegador para obter o Beta by Crashlytics.

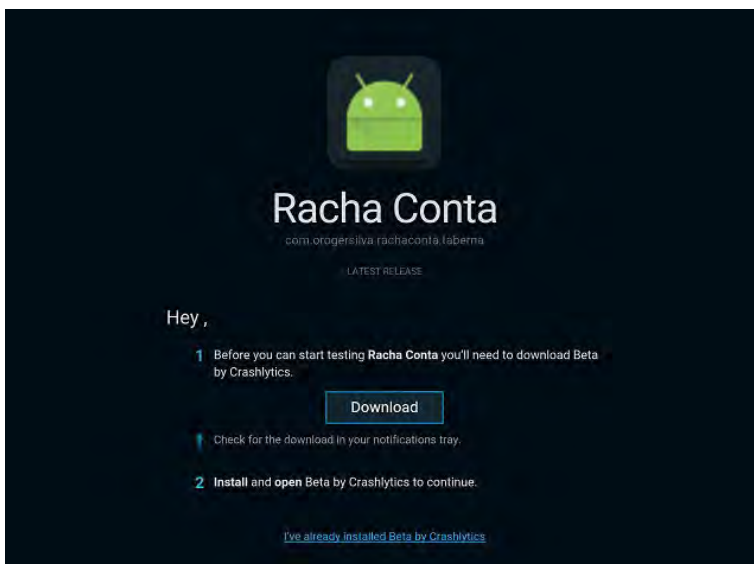


Figura 5.34: Download do Beta by Crashlytics

Uma vez instalado o Beta by Crashlytics no dispositivo mobile do usuário, ele terá acesso ao app distribuído.

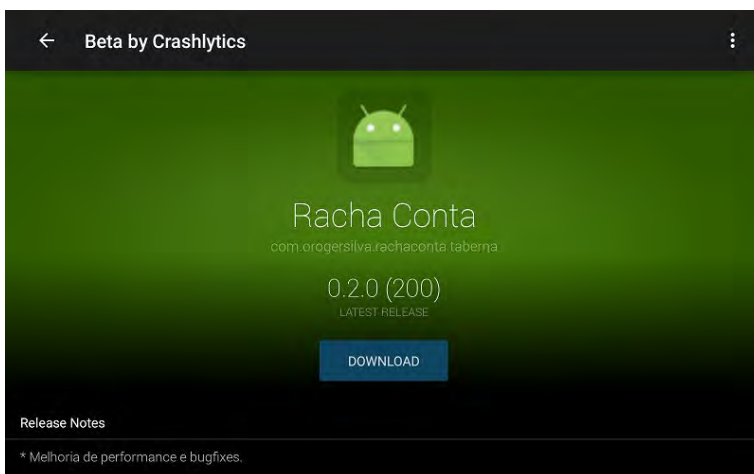


Figura 5.35: Download do app via Beta by Crashlytics

Distribuindo apps via Crashlytics (com CI)

Nossa empresa faz, em média, dez deploys por dia para grupos de usuários. O processo ainda não é automatizado, mas, em breve, será.. Pobre do responsável por essa tarefa. Relembrando o processo de distribuição: gerar a nova versão do app; arrastá-la até o painel do plugin do Crashlytics; selecionar os usuários-alvo; e, finalmente, despachar o app. Isso, dez vezes ao dia. 50 vezes toda semana. 200 vezes por mês. É um absurdo!

Tenha sempre em mente: máquinas são excelentes para realizarem tarefas repetitivas. Então, vamos automatizar as distribuições!

Para exemplificar a automatização de uma distribuição via Crashlytics, vamos configurá-la para distribuir apps para um grupo de usuários (assim como exemplificado na seção sobre HockeyApp). Então, um grupo de usuários será criado no dashboard do Beta by Crashlytics. Ele será nomeado `Racha Conta Testers` e um usuário adicionado a ele. Contudo, o grupo poderia ter dezenas de usuários, caso necessário, de modo que todos eles seriam notificados a cada nova versão distribuída para o grupo.

Um novo diretório será criado na raiz do projeto do app no Android Studio e será nomeado `crashlytics`. Esse diretório conterá informações para viabilizar a distribuição do app via Crashlytics de forma automatizada.

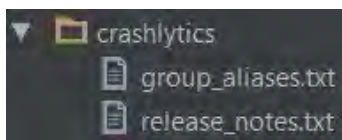


Figura 5.36: Definindo configurações para distribuição automatizada via Crashlytics

O arquivo `release_notes.txt` contém as novidades na nova versão do app a ser distribuída. A cada nova distribuição ele deverá ser atualizado (caso necessário). Já o arquivo `group_aliases.txt`

conterá os identificadores dos grupos de usuários (separados por vírgula) que receberão a distribuição. Esse identificador trata-se de um *alias*, que pode ser obtido através do dashboard do Beta by Crashlytics, na opção de menu `Manage Groups` .



Figura 5.37: Alias do grupo de usuários

Também no arquivo `build.gradle` . em nível de módulo, devem ser apontados os arquivos `group_aliases.txt` e `release_notes.txt` , para que a task Gradle de distribuição conheça as configurações desse processo de distribuição. O seguinte trecho de código deve ser adicionado ao corpo do *build type* de interesse:

```
ext.betaDistributionGroupAliasesFilePath="${rootProject.getRo
otDir()}/crashlytics/group_aliases.txt"
ext.betaDistributionReleaseNotesFilePath="${rootProject.getRo
otDir()}/crashlytics/release_notes.txt"
```

Assim como realizado para o HockeyApp, será configurado um job no Jenkins para deployar novas distribuições para grupos de usuários. O job contém todas as configurações iguais ao job

`RachaContaAndroidPublishGooglePlay` , com a exceção da configuração da task, que deve ser executada durante o build. Neste caso, será a task `crashlyticsUploadDistributionRelease` .

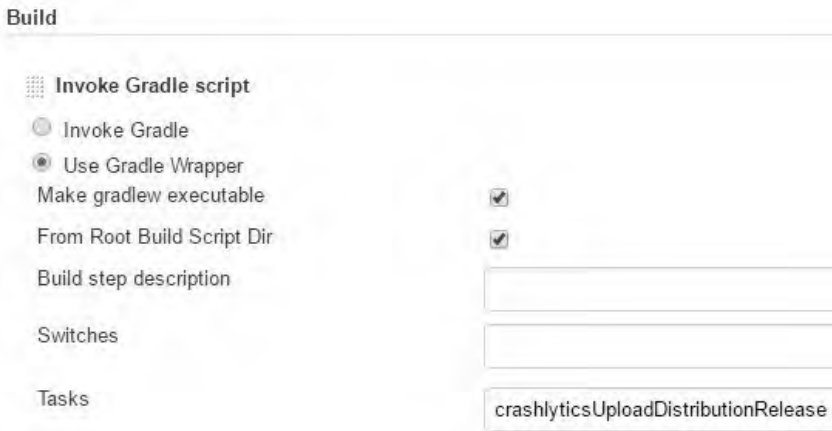


Figura 5.38: Configuração da task de publicação do Crashlytics

Deste modo, a cada nova alteração enviada sobre o projeto do app para o repositório de código remoto, ao final do pipeline de deployment, um estágio de nome `RachaContaAndroidPublishCrashlytics` será responsável por distribuir o app via Crashlytics.

5.5 CONCLUSÃO

Se você chegou aqui, parabéns! Você sabe implementar um pipeline de deployment para apps Android. Entrega contínua? Depende. Trata-se uma prática. O time de desenvolvimento deve querer levá-la ao pé da letra em seu dia a dia.

O build do app foi quebrado? Por acaso o time dedicou toda sua atenção em consertar o build e "rebuildar" a aplicação? O build terminou com sucesso e foi decidido publicar o app para o Google Play? E toda vez que ocorre uma quebra, esses passos são repetidos?

E se uma nova funcionalidade é agregada ao app, testes automatizados são escritos para cobrir esse cenário com um novo build sendo processado através do pipeline, como forma a detectar

uma possível regressão? O build foi um sucesso novamente? E foi decidido já publicar o artefato resultante desse build? E a postura do time é a mesma várias vezes ao dia durante todo o ciclo de vida da aplicação?

Meus parabéns! Bem-vindo à entrega contínua em Android. :)

BIBLIOGRAFIA

ANICHE, M. *Test-Driven Development: Teste e Design no Mundo Real*. São Paulo: Casa do Código, 2012.

BAPTISTA, V. *Continuous Integration for Android Apps with Jenkins and Maven3*. Jul. 2011. Disponível em: <http://vitorbaptista.com/continuous-integration-for-android-apps-with-jenkins-and-maven3/>. Acessado em: 14 fev. 2016.

BRISON, V. *How to Improve Quality and Syntax of Your Android Code*. Jul. 2014. Disponível em: <http://vincentbrison.com/2014/07/19/how-to-improve-quality-and-syntax-of-your-android-code/>. Acessado em: 29 abr. 2016.

CADET, A. *Android Continuous Integration Using Gradle, Android Studio and Jenkins*. Mar. 2015. Disponível em: <https://www.coshx.com/blog/2015/03/31/android-continuous-integration-using-gradle-android-studio-and-jenkins/>. Acessado em: 17 abr. 2016.

COHN, M. *User Stories Applied: For Agile Software Development*. Boston: Pearson Education, 2004.

COMPTON, M. *Continuous Delivery for Android*. Fev. 2015. Disponível em: <https://www.bignerdranch.com/blog/continuous-delivery-for-android/>. Acessado em: 28 mar. 2016.

CONTINUOUSAGILE.COM. *Continuous Stories - Mobile App*.

Disponível em:
http://www.continuousagile.com/unblock/cd_mobile.html.
Acessado em: 02 mar. 2016.

DRIESSEN, V. *A Successful Git Branching Model*. Jan. 2010. Disponível em: <http://nvie.com/posts/a-successful-git-branching-model/>. Acessado em: 03 abr. 2016.

DUVALL, P. M.; MATYAS S.; GLOVER, A. *Continuous Integration: Improving Software Quality and Reducing Risk*. Boston: Pearson Education, 2007.

HAMBRICK, R. *Continuous Delivery for Android (Part 1)*. Dez. 2014. Disponível em: <http://blog.stablekernel.com/continuous-delivery-android-part-1/>. Acessado em: 15 mar. 2016.

HUMBLE, J.; FARLEY, D. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Boston: Pearson Education, 2010.

MAÑAS, E. L. *Automating Android Development*. Mai. 2015. Disponível em: <https://medium.com/google-developer-experts/automating-android-development-6daca3a98396#.d57bdpfro>. Acessado em: 23 abr. 2016.

MARTINEZ, J. *Use GoCD for Android and Get Rid of Jenkins*. Abr. 2016. Disponível em: http://jeremie-martinez.com/2016/04/19/gocd-android/?utm_source=androiddevdigest. Acessado em: 11 jun. 2016.

ROMER, T. *Using Jenkins to Build Pipelines, Chain and Visualize Jobs*. Maio 2014. Disponível em: <http://zeroturnaround.com/rebellabs/how-to-use-jenkins-for-job-chaining-and-visualizations/#outofthebox>. Acessado em: 15 abr. 2016.

SWANSON, M. *Integration Testing against REST APIs in Android*. Fev. 2014. Disponível em: <http://mdswanson.com/blog/2014/02/24/integration-testing-rest-apis-for-android.html>.